NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A STATIC SCHEDULER FOR CRITICAL TIMING CONSTRAINTS

by

Laura C. Marlowe

December 1988

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified		1b Restrictive Markings	
2a Security Classification Authority		3 Distribution Availability of Report: Approved for public release; distribution is unlimited.	
2b Declassification/Downgrading Schedule			
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)	
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (*If Applicable*): 37	7a Name of Monitoring Organization: Naval Postgraduate School	
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000		7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	
8a Name of Funding/Sponsoring Organization	8b Office Symbol (*If Applicable*)	9 Procurement Instrument Identification Number	
8c Address (*city, state, and ZIP code*)		10 Source of Funding Numbers: Program Element Number / Project No / Task No / Work Unit Accession No	

11 Title (*Include Security Classification*): A Static Scheduler for Critical Timing Constraints

12 Personal Author(s): Laura C. Marlowe

13a Type of Report	13b Time Covered	14 Date of Report (*year, month,day*)	15 Page Count
Master's Thesis	From To	December 1988	155

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	Rapid prototyping, Static scheduler, CAPS, PSDL, Ada, Computer aided prototyping, Kodiyak, Time-critical, Hard real-time constraints

19 Abstract (*continue on reverse if necessary and identify by block number*

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) represent a pioneering effort in the field of software development. The implementation of CAPS will enable software engineers to automatically validate design specifications and functional requirements early in the design of a software system through the development and execution of a prototype of the system under construction.

Execution of the prototype is controlled by an Execution Support System (ESS) within the framework of CAPS. One of the critical elements of the ESS is the Static Scheduler which extracts critical timing constraints and precedence information about operators from the PSDL source that describes the prototype. The Static Scheduler then uses this information to determine whether a feasible schedule can be built, and if it can, constructs the schedule for operator execution within the prototype.

20 Distribution/Availability of Abstract: [X] unclassified/unlimited [ ] same as report [ ] DTIC users	21 Abstract Security Classification: Unclassified	
22a Name of Responsible Individual: LuQi	22b Telephone (*Include Area code*): (408) 646-2735	22c Office SymbolT.: 52Lq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted

All other editions are obsolete

security classification of this page: Unclassified

Approved for public release; distribution is unlimited

A Static Scheduler for Critical Timing Constraints

by

Laura C. Marlowe
Lieutenant Commander, United States Navy
B.A., Rollins College, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) represent a pioneering effort in the field of software development. The implementation of CAPS will enable software engineers to automatically validate design specifications and functional requirements early in the design of a software system through the development and execution of a prototype of the system under construction.

Execution of the prototype is controlled by an Execution Support System (ESS) within the framework of CAPS. One of the critical elements of the ESS is the Static Scheduler which extracts critical timing constraints and precedence information about operators from the PSDL source that describes the prototype. The Static Scheduler then uses this information to determine whether a feasible schedule can be built, and if it can, constructs the schedule for operator execution within the prototype.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. THE COMPUTER AIDED PROTOTYPING SYSTEM PROCESS 1

B. THE STATIC SCHEDULER 4

C. ORGANIZATION 5

II. THE COMPUTER AIDED PROTOTYPING SYSTEM ENVIRONMENT 6

A. PROTOTYPING SYSTEM DESCRIPTION LANGUAGE 6

1. PSDL's Purpose 6

2. The Computational Model 6

3. Operators and Data Streams 7

4. Control Constraints 9

5. Timing Constraints 10

B. SEMANTICS-RELATED HARD REAL-TIME CONSTRAINTS 11

C. THE EXECUTION SUPPORT SYSTEM 13

D. THE FUNCTION OF THE STATIC SCHEDULER 14

E. ARCHITECTURAL DESIGN 16

III. ELEMENTS OF THE STATIC SCHEDULER 18

A. THE KODIYAK PROGRAMMING LANGUAGE 18

B. THE PROGRAMMING LANGUAGE ADA 21

IV. DEVIATIONS FROM THE ORIGINAL DESIGN 26

A. DATA STRUCTURES 26

1. The N-ary Tree 28

2. Linked Lists 31

3. Variable Length Strings 34

B. EXCEPTION HANDLING 35

C. PACKAGE IMPLEMENTATION 37

1. The FILES Package....37
2. The PSDL_READER Package....38
3. The FILE_PROCESSOR Package....39
4. The TOPOLOGICAL_SORTER Package....44

V. DIFFICULTIES WITH THE DESIGN....47
A. OPERATOR'S AND LINK STATEMENT'S METS....47
B. COMPOSITE AND ATOMIC LINK STATEMENTS....48
C. NON-TIME CRITICAL OPERATORS AND PRECEDENCE....52
D. DECOMPOSITION INTO LINEAR AND NETWORK-LIKE GRAPHS....55
E. VALIDATION OF OTHER TIMING CONSTRAINTS....58
F. ATOMIC OPERATOR NAMING CONVENTIONS....60

VI. CONCLUSION....63

LIST OF REFERENCES....66

APPENDIX A. PROTOTYPE SYSTEM DESCRIPTION LANGUAGE GRAMMAR....68

APPENDIX B. ATTRIBUTE GRAMMAR SOURCE CODE....72

APPENDIX C. IMPLEMENTATIONS OF THE STATIC SCHEDULER'S ABSTRACT DATA TYPES....86

APPENDIX D. STATIC SCHEDULER SOURCE CODE....123

BIBLIOGRAPHY....142

INITIAL DISTRIBUTION LIST....143

LIST OF TABLES

TABLE 1. DATA TYPES AND THEIR CORRESPONDING DATA STRUCTURES 27
TABLE 2. RECORD FIELDS FOR OPERATORS 28
TABLE 3. RECORD FIELDS FOR LINKS 31
TABLE 4. RECORD FIELDS OF PRECEDENCE 33

LIST OF FIGURES

Figure 1. Computer Aided Prototyping System Overview 3

Figure 2. A Network Decomposition of a Composite Operator with Associated Link Statements 9

Figure 3. Architectural Design of the Static Scheduler 17

Figure 4. PSDL Operator Decomposition in an N_ary Tree 30

Figure 5. Three Representations of a Single Link Statement 32

Figure 6. Composite and Atomic Operators in the Operator Tree Data Structure 44

Figure 7. A PSDL Graph and Associated Link Statements 46

Figure 8. Composite to Atomic Relationships in a Decomposition 50

Figure 9. Original Graph and Link Statements 53

Figure 10. Linear Versus Network Decompositions 57

Figure 11. Path Construction from a Network 59

Figure 12. Atomic Operator's Module Used Twice 62

I. INTRODUCTION

A. THE COMPUTER AIDED PROTOTYPING SYSTEM PROCESS

Advances in software design methodology have not kept pace with the rapid advances in computer hardware leaving a gap between the capabilities of the hardware that is currently available and the software systems that can be designed and implemented to exploit the hardware's full capabilities. This is especially true in the large embedded systems, requiring real-time information and data processing, that are being designed for the Department of Defense (DOD). In a world where the demand for complex software systems has increased faster than the technology necessary to build these systems, a new approach is needed to speed up the software engineer's ability to design, test and implement large, complex software systems with high degrees of reliability.

The Computer Aided Prototyping System (CAPS) process is one proposed method for speeding up the design and implementation of large software systems while increasing the reliability of the final product and, at the same time, reducing the need for expensive design changes during the latter stages of software development. This process utilizes an approach called rapid prototyping combined with a reusable software management base to produce a prototype of the system being designed. A prototype is an executable model of the intended system, while rapid prototyping is the construction activity leading to the prototype. The software base is a data base containing reusable software modules in the high-level computer language the system is being written in. These reusable modules will form much of the body of the prototype being constructed, saving many man hours

in the construction process. The executable model allows the designer to test the system and verify that it meets the needs of the user, and is feasible within the requirements specified by the user. [Ref. 1]

The CAPS process is an iterative approach to designing complex software systems. Figure 1 represents an overview of CAPS as viewed by this thesis. The designer using CAPS enters specifications for the entire system in the Prototype System Description Language (PSDL) and then, the sequence control function of the CAPS User Interface [Ref. 2] requests that the Software Base Management System [Ref. 3] initiate a search of the reusable software base for component modules that meet these specifications. If found, they are retrieved and the designer is allowed to choose which one of them to include as the prototype. If a suitable match cannot be found between the specifications for a module and the software components in the software base, the system must be decomposed into its component parts. Using the editing tools provided within CAPS, the system designer decomposes the system into subsystems, each having their own specifications. The software base is searched for component modules that meet the specifications for the subsystem. If found, they are retrieved and become part of the prototype. This process of decomposition and searching the software base is continued until the point is reached in each subsystem where decomposition is no longer possible, or is not feasible due to the simplicity of the subsystem. When this point is reached, the module must be written in the base language in which the system is being implemented and included in the prototype.

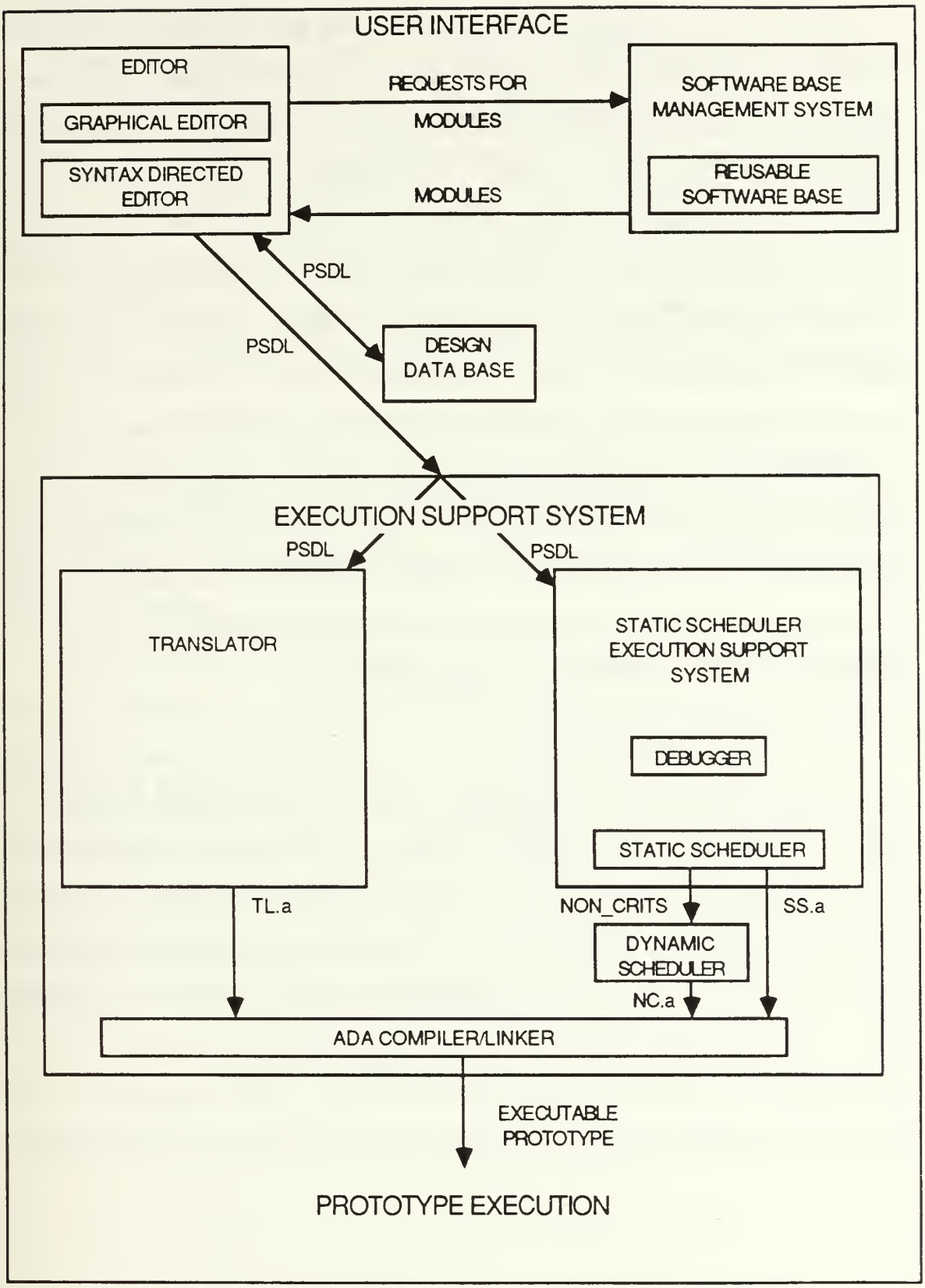


Figure 1. Computer Aided Prototyping System Overview

Once all of the specifications have been met and appropriate software components composed into the proposed prototype, the User Interface sequence control function passes the PSDL description of the prototype to the Execution Support System where it is executed. The designer and the user examine the behavior of the prototype during execution to ensure that the requirements specified by the user are met, and that the system will do what is actually intended. Requirements are adjusted and the prototype is modified until the customer and designer are satisfied that the prototype performs correctly. The last step in the process is to build the final system from the framework provided by the prototype. This last step need only be done once since the system design has been verified through the prototype. Implementation should proceed rapidly because the skeleton of the system has already been designed and tested and contains many of its actual components. The prototype in the base language need only be fleshed out to perform the true tasks of the software system. [Ref. 1]

B. THE STATIC SCHEDULER

One of the most critical components of the Computer Aided Prototyping System (CAPS) is the Static Scheduler subsystem of the Execution Support System (ESS). This thesis describes the initial efforts to implement the Static Scheduler according to the original design [Ref. 1]. This design was further elaborated in more detail in [Ref. 4] and in the implementation guide for the Static Scheduler [Ref. 5]. The algorithms used in the actual implementation follow as closely as possible those given in the implementation guide. Where clarification was necessary, the detailed design of O'Hern [Ref. 4] and Luqi [Ref. 1] were consulted.

C. ORGANIZATION

Chapter II presents the CAPS environment in which the Static Scheduler will function. It includes a discussion of the Prototyping System Description Language (PSDL) as it relates to the Static Scheduler and the semantics-related hard real-time constraints with which the Scheduler must deal. The Static Scheduler's role within the Execution Support System (ESS) is described, as is its function. The architectural design of the Static Scheduler is presented as a background for the rest of this thesis. The design implemented is given by O'Hern [Ref. 4] and Janson [Ref. 5].

Chapter III thoroughly discusses the two languages used in the implementation of the Static Scheduler, describes their separate roles and the reason two languages were necessary. These languages are Kodiyak and Ada, the Department of Defense's language.

Chapter IV presents the completed portion of implementation of the Static Scheduler and gives a thorough explanation of the deviations from the original design that were necessary in order to make the implementation work.

Chapter V describes and discusses the problems that this design presents to the implementation and offers some possible solutions to these problems. In fact, solutions to the problems discovered in this initial effort must be found and implemented within this partial implementation before the Static Scheduler can be completed.

II. THE COMPUTER AIDED PROTOTYPING SYSTEM ENVIRONMENT

A. PROTOTYPING SYSTEM DESCRIPTION LANGUAGE

1. PSDL's Purpose

The Prototype System Description Language (PSDL) was designed for describing real time software systems. It can be used for requirements analysis and feasibility studies for software systems, and, ultimately, for the design of large embedded systems. PSDL has features for describing critical timing constraints, control constraints, and abstractions for operators and data within the system being designed. It was designed for use within the Computer Aided Prototyping System (CAPS) and depends on CAPS as its support environment.[Ref. 6]

Although PSDL has many features necessary to the construction of a prototype software system, only those that relate to the Static Scheduler will be described here. These features include operator abstractions, data streams and critical timing constraints and control constraints. A more thorough description of all of the features of PSDL can be found in [Ref. 1] and [Ref. 6].

2. The Computational Model

PSDL is based on a computational model containing operators that communicate via data streams. An operator may be sporadic, that is data driven, or periodic. Data streams carry values of data between operators. The formal computational model is a graph given by

$$G = (V, E, T(v), C(v))$$

where G is the graph, V is the set of vertices within the graph, E is the set of edges connecting the vertices, T(v) is the maximum execution time (MET) for each vertex v, and C(v) is the set of control constraints for each vertex v. In PSDL each vertex in the graph is an operator, which may have a maximum execution time, and each edge in the graph is a data stream. [Ref. 6] These PSDL decomposition graphs determine the semantics of a PSDL system when combined with the semantics of the operators that appear in the graph. [Ref. 7]

3. Operators and Data Streams

An operator within PSDL may be either a function or a state machine. When an operator executes it reads a data value from each of its input data streams and may produce a data value on each of its output data streams. If the operator is a function, its outputs depend only on its input values. If the operator is a state machine its outputs depend both on its input values and its internal state variables.

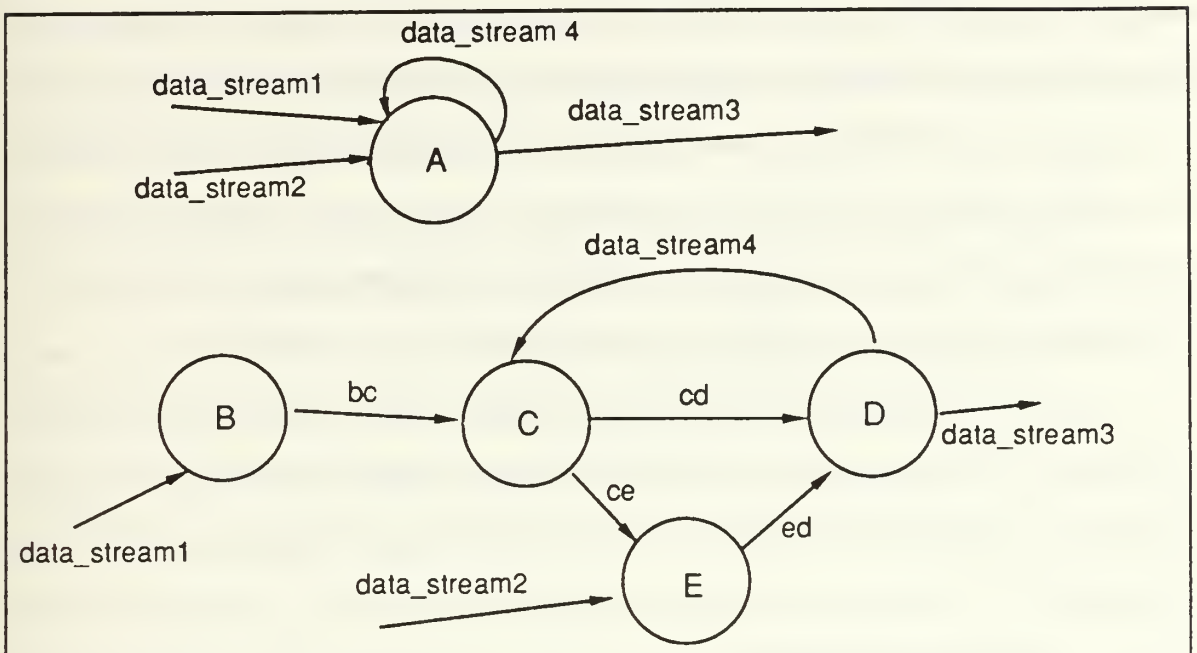
Regardless of whether an operator is a function or a state machine, an operator may be composite or atomic. Atomic operators cannot be further decomposed into other operators while composite operators may be decomposed into a network of lower level operators with their connecting data streams. Figure 2 shows a composite operator A and its network decomposition into operators B, C, D and E. If two operators in a decomposition are connected by a data stream, then there is an explicit precedence relationship between them. For instance, in Figure 2 the output of operator B is an input to operator C, so these two operators have a precedence relationship. Operator B must be scheduled to execute before operator C. A composite operator whose network decomposition contains cycles is a state machine. In a state machine one, or more, of the operator's data streams is designated as a state variable. A state variable is both an output and input feedback

loop within the operator and is always given an initial value. Data_stream4 in Figure 2 is an example of a state variable. This state variable serves to allow the circular precedence relationships among operators to be broken without destroying the semantics of the network. As can be seen in Figure 2 data streams are the communication links between operators. A data stream connects exactly two operators, the producer of the data stream, and the consumer of the data stream. [Ref. 6]

PSDL operators have two parts, the specification of the operator and the implementation of the operator. The specification contains attributes describing the interface, timing characteristics and the behavior of the operator. The attributes specify both the operator and form the basis for retrieval of reusable modules for their implementation from the software base. [Ref. 6] Within the specification of an operator, the Static Scheduler is only interested in the state variable declarations, the timing constraints and some of the control constraints for the operator.

The implementation of an operator determines whether the operator is atomic or composite. If the operator is atomic, this part will contain a keyword specifying the underlying language in which the prototype is being built. In our case, this language is Ada. [Ref. 6] If the operator is composite, this part will contain the keyword "GRAPH" followed by the set of "link statements" that represent the graphic decomposition of the operator into its components. A link statement is used to describe the relationship between two operators by indicating the direction of flow of the data stream between them. This is depicted in the graph in Figure 2 by the arrowhead on the edges representing the data streams. A link statement is read as data_stream.output_operator -> input_operator. [Ref. 4] For

instance, the first link statement in Figure 2 says that data_stream1 is an output from an external operator and is connected as an input to operator B.



data_stream1.EXTERNAL → B
bc.B → C
cd.C→ D
data_stream 4.D → C
ce.C → E
data_stream2.EXTERNAL → E
ed.E → D
data_stream3.D → EXTERNAL

Figure 2. A Network Decomposition of a Composite Operator with Associated Link Statements

4. Control Constraints

There are many aspects of control constraints for PSDL operators. Of these, the Static Scheduler is interested in only two. These are the PERIOD and FINISH_WITHIN constraints. The PERIOD is the length of time between the start

of the scheduling interval and the start of the next scheduling interval for an operator [Ref. 7]. An operator may be periodic or sporadic. A periodic operator must be scheduled at regular time intervals. Within the period specified, the operator must be scheduled to execute and complete execution. Sporadic operators are triggered by the arrival of new data values on their input streams and will have irregular time intervals between their execution. An operator is periodic if it has a period specified or is a component of a periodic composite operator from which it will inherit its period. An operator that is not periodic is sporadic.[Ref. 1]

The FINISH_WITHIN constraint is optional on a periodic operator and specifies the time interval in which an operator must finish execution. The FINISH_WITHIN is the deadline for the scheduling interval of a periodic operator.[Ref. 7]

5. Timing Constraints

PSDL's essential timing constraints are given in the specification for an operator and consist of the maximum execution time (MET), maximum response time (MRT) and minimum calling period (MCP). These timing constraints are important in specifying the hard real-time requirements often found in embedded systems.

The maximum execution time (MET) gives the upper limit on the length of time between the moment an operator begins execution and the moment it completes execution. An MET may be specified for any operator within the prototype and is required for every operator that has any timing constraints.

For a sporadic operator, the maximum response time (MRT) is an upper bound on the time between the arrival of a new set of input data values to a sporadic operator, and the time when the last of its output values is put into its output data

streams. The minimum calling period (MCP) of a sporadic operator is a constraint on its environment which specifies a lower bound between the arrival of one set of inputs and the arrival of the next set of inputs.[Ref. 6] In a PSDL specification, every sporadic operator that has timing constraints must have a MRT time constraint and a MCP time constraint in addition to an MET [Ref. 7].

B. SEMANTICS-RELATED HARD REAL-TIME CONSTRAINTS

One of the goals of PSDL is to simplify the design of systems with hard real-time constraints. Many times, the need for meeting real-time deadlines results is designs where conceptually unrelated tasks must be interleaved, making their implementations hard to understand. PSDL deals with this problem by presenting a high-level description of the system in terms of networks of independent operators, while allowing the interleaving of separate tasks to be handled by an automatic translator that generates lower level code. High-level synchronization is handled by using dataflow streams to coordinate the arrival of various data values from different sources. The Static Scheduler for the time-critical operators eliminates the need for other kinds of synchronization by the system designer.[Ref. 7]

The timing and control constraints described in the previous section must have additional relationships with one another in order for a valid schedule to be feasible. An operator that has any timing constraints must have an MET in order for the Static Scheduler to build a schedule. If an operator has a MCP value, then it must also have a MRT value. When a MRT value is present, its value must be greater than the MET since the execution time for an operator must be less than the response time to accomplish the operator's tasks prior to the time when the output is required. If only a single processor is available, then, for periodic operators, the

PERIOD must be greater than the MET so that the operator can never be required to be scheduled to execute before the previous execution is complete.[Ref. 4]

Sporadic operators are implemented in the Static Scheduler by calculating their periodic equivalents. The period of an equivalent periodic operator is found by use of the following formula.

PERIOD = minimum(MCP, MRT - MET)

An equivalent periodic operator derived in this manner has a deadline that is equal to the maximum execution time. This means that the operator must be scheduled to start at the beginning of each period because it cannot meet its timing constraints unless the period is greater then the maximum execution time of the operator.[Ref. 6]

In the decomposition process of an operator, its network of component operators must all be able to complete execution within the time constraints of the composite. When a composite operator has a MET, all of its component operators must have a MET to ensure completion of execution of the decomposition prior to the time scheduled for completion of the composite operator. The sum of the METs along each and every data stream path within the network must be less than or equal to the MET of the composite. This also implies that every operator within a decomposition must have a MET less than or equal to the MET of the composite.

In addition to the timing and control constraints on operators, there is an implicit constraint on their scheduling. This is the dataflow precedence constraint which is derived from the network graph that represents an operator's decomposition. Since this graph is a directed graph, there is a dataflow ordering specified by the data streams that connect the operators one to another. Once state variables are removed, the graph becomes acyclic. The result is that there is a strict

partial ordering within the graph that determines the order in which operators must be scheduled to execute.[Ref.7] It can be seen in Figure 2 that after the edge data_stream4, a state variable, is removed, that operator B must be scheduled first, followed by operators C, E and D in succession. Any other ordering within the schedule would result in an operator attempting to execute without the proper inputs.

C. THE EXECUTION SUPPORT SYSTEM

PSDL prototypes are executable if all required information is supplied, and the software base contains implementations for all atomic operators and types. The execution and testing of the prototype depends upon a subsystem of CAPS known as the Execution Support System (ESS). The ESS contains a Static Scheduler, a Translator, and a Dynamic Scheduler.[Ref.1] The purpose of the ESS is to turn the PSDL description of the system under construction into an executable prototype using the software components that have been retrieved from the software base or written for the prototype. During this process, validation of the critical timing information provided by the designer is done, control constraints are translated into the base language of the system, and the base language modules are organized for final execution. Once this has been done, the prototype is executed.

Briefly, the Static Scheduler attempts to find a static schedule for all operators with real-time constraints. The Static Scheduler is the subject of this thesis and will be discussed in more detail in the following section.

The Translator augments the implementations of the atomic operators and types with code in the base language of the prototype that realizes the data streams and control conditions, resulting in a program that can be compiled and executed. The augmentations it produces serve to adapt the atomic operators to the context in

which they are going to operate. The Translator implements four PSDL constructs that are necessary to tie the operators within the prototype together. These are data streams, conditionals, timers and exceptions. The resulting code is an Ada program that simulates the behavior of the PSDL prototype. This code is then compiled with code from the Static Scheduler and Dynamic Scheduler and may then be executed.[Ref. 1] The current implementation of the Translator can be found in Altizer's thesis [Ref. 8] which gives an in depth discussion of the PSDL constructs that have been implemented.

The Dynamic Scheduler is the runtime executive for the Execution Support System. Its main function is to execute the prototype. It does this by invoking the time critical operators in the order specified by the Static Schedule, and then using remaining time slots between completion of one time critical operator and the scheduled start of the next time critical operator, to execute non-time critical operators on the processor.[Ref. 1] It also has debugging facilities which are used to pass information to the prototype designer about errors discovered in the construction of the Static Schedule, and runtime errors during execution of the prototype. More information and the current implementation of the Dynamic Scheduler can be found in Wood's thesis.[Ref. 9]

D. THE FUNCTION OF THE STATIC SCHEDULER

The function of the static scheduler is to build a static schedule for the execution of the prototype being developed from the PSDL input specification for the prototype. This schedule gives the precise execution order and timing of operators with hard real-time constraints in such a manner that all timing constraints are guaranteed to be met.[Ref. 4]

The Static Scheduler does this by analyzing the real-time constraints given in the PSDL prototype and attempting to find a static schedule that meets the timing constraints of the time critical operators. An operator is considered time critical if it has at least one hard real-time constraint which must be met. The final Static Schedule specifies, in advance of execution, time slots that are allocated for each critical operator that are sufficiently long for their worst case execution time.[Ref. 1]

When designing prototypes for hard real-time systems, the timing and control constraints for operator firing and execution are critical. In the computational model of the prototype, each time critical operator includes a maximum execution time which gives the worst case time to complete execution once the operator fires. Critical operators can also include condition control constraints. These constraints stipulate firing conditions for an operator, and conditions necessary before an operator produces output or exceptions.

PSDL control abstractions provide a means to explicitly describe the periodic execution of operators. These abstractions are represented as a set of control constraints within the PSDL specification of each operator. The actual order of operator execution is determined by the Static Scheduler. The Scheduler utilizes these constraints to recognize the precedence relationships between the data flow diagrams of the operators.

In order to build a schedule from the PSDL input, it is necessary to extract the timing, control and precedence relationship information from the PSDL input. This information is then used to schedule the PSDL operators for execution of the prototype.[Ref. 4]

Within the Static Scheduler, sporadic operators are implemented using their periodic equivalents. The Static Scheduler then partitions the set of periodic operators into non-overlapping harmonic blocks, one for each processor available. In this implementation only a single harmonic block will be built since currently only a single processor system is available. A harmonic block has two properties. The periods of all of the operators in the block are exact multiples of the base period of the block, and at least one of the operators in the block has a period equal to the base period.

The Static Schedule is a table that gives the starting times and execution times for each operator in the harmonic block, and covers a length of time equal to the least common multiple of all of the periods in the block.[Ref.1]

E. ARCHITECTURAL DESIGN

The architectural design used in this implementation was originally described in Luqi's doctoral dissertation, *Rapid Prototyping for Large Software System Design* [Ref. 1] and was further developed by O'Hern [Ref. 4] and Janson [Ref. 5]. Figure 3 is a representation of the top level data flow diagram of the design and can be found in both [Ref. 4] and [Ref. 5].

In this design the first module, known as PSDL_READER, reads in the PSDL source file for the prototype being designed and produces a text file containing only the information required by the Static and Dynamic Schedulers.

The resultant text file becomes the input to the module FILE_PROCESSOR. This module's function is to separate the text file into three separate files that will be further processed by the Static Scheduler. These files are the Operator File, referred to as OPERATORS, which contains all of the operators with their time critical information, the Links File, referred to as LINKS, which contains

precedence information about the operators, and the Non Critical File, referred to as NON_CRITS, which contains operators whose scheduling and execution is not time critical and is used later by the Dynamic Scheduler which schedules non-time critical operators for execution. All other information that may still be in the text file will be ignored.

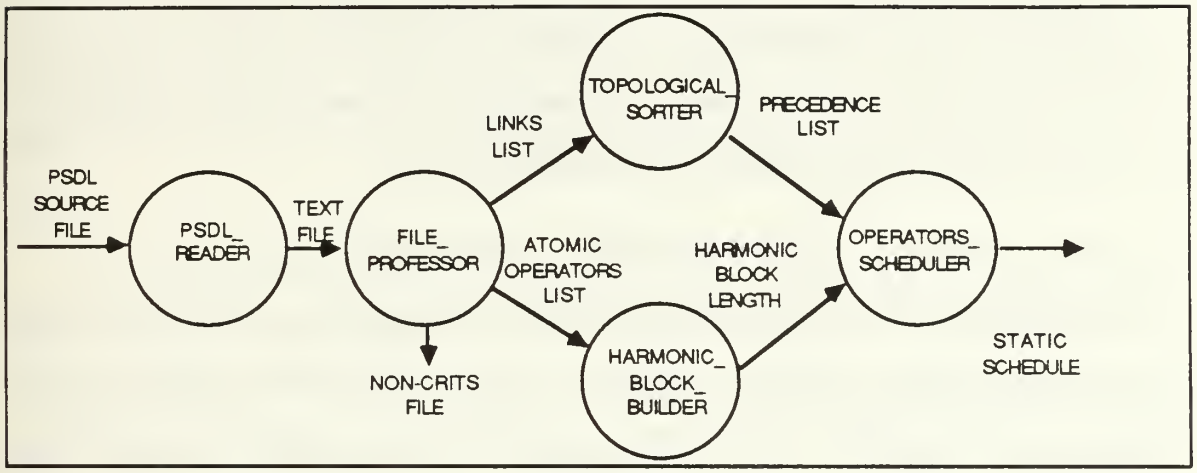


Figure 3. Architectural Design of the Static Scheduler

The module HARMONIC_BLOCK_BUILDER creates an harmonic block template that is tailored to the critical operators and their firing intervals. These blocks are based on the minimum calling period (MCP), the minimum response time (MRT) and the maximum execution time (MET) of each operator.

The TOPOLOGICAL_SORTER module's function is to take the link information about the operators and build precedence relationships which specify which operators must complete execution before other operators can execute.

Finally, the OPERATOR_SCHEDULER module combines the precedence list and the harmonic block length information produced by the previous two modules to produce the final Static Schedule of operators.

III. ELEMENTS OF THE STATIC SCHEDULER

The Static Scheduler consists of two components which have been built using two different language tools. The first is Kodiyak, a language that was designed for construction of translators based on attribute grammars. The second is Ada, which was designed for the Department of Defense for use in embedded computer systems, as well as for use as a general purpose language. Both of these languages will be discussed in further detail in the following sections.

A. THE KODIYAK PROGRAMMING LANGUAGE

The language Kodiyak was designed for the construction of translators for context-free attribute grammars and was based upon Knuth's description of attributed grammars [Ref. 10]. Attribute grammars are a method for describing syntax-directed translation, where the context-free grammar's symbols are augmented with attributes and the context-free grammar's rules are augmented with equations which define those attributes. The Kodiyak compiler accepts a context-free grammar, attribute declarations and equations, a scanner specification and output declarations for the language for which a translator is to be generated and constructs the described translator.

A Kodiyak program is divided into three sections separated from each other by a double percent symbol on a line by itself. The first section describes what will become the features of the lexical scanner which is used to parse the source text into tokens, and gives operator precedence for the tokens. The lexical scanner of the Kodiyak program defines a set of substitutions to be performed on the text that is input to it. This is done using named terminal symbols which are associated with

regular expressions. The input is scanned for sections of text which match these regular expressions and when found, each regular expression is replaced by its associated named terminal symbol.

The second section gives a name to the attributes which are associated with each grammar symbol in the language for which a translator is being generated, and declares their types. These types may be either strings or integers. Strings may be of arbitrary length and may be concatenated together. The integer range depends upon the machine being used, and mathematical operations may be performed on them.

The most difficult aspect of using Kodiyak is writing the third section which defines the syntax and semantics of the attribute grammar of the language for which a translator is being generated. It consists of a set of Backus-Nauer Form (BNF) equations that define the grammar rules of the language. The equations consist of a non-terminal on the left followed by a colon and a series of terminals and non-terminals that represent the grammar rules. Attribute equations for the syntax follow, surrounded by curly braces. These equations cause the input program in the source language to be correctly and unambiguously parsed into tokens that can then be translated by the use of the attribute equations into either terminals or non-terminals. Non- terminals are further parsed until only terminals are used in their decomposition. Once only terminals are produced, the translator will output one of the two primitive data types, strings or integers, depending upon the attribute type declared previously for the terminal.[Ref. 11]

The PSDL translator, or attribute grammar processor, produced for the Static Scheduler is based on preliminary work done by Janson [Ref. 5] and Moffitt [Ref. 12]. The complete BNF description of the PSDL translator as it appears

without the attribute translations was done in conjunction with Altizer [Ref. 8] since both Altizer's PSDL Translator and this thesis depend upon the correct parsing of the PSDL grammar. The final translations, or outputs from the original translator were then developed individually into separate translators since the output desired in both cases is completely different. The PSDL Translator developed by Altizer translates PSDL source code into compilable Ada source code, while the attribute grammar processor developed for this thesis, produces a text file of operator names, keywords and critical timing information and precedence relation information about the operators.

The Static Scheduler uses Kodiyak to build an attribute grammar processor for the first module of the Static Scheduler, which is known as the PSDL_READER module in the implementation guide written by Janson [Ref. 5]. The function of this module is to parse the original input PSDL program and generate the input required by the second module, which is known as FILE_PROCESSOR. In order to do this, the translator that is generated by Kodiyak recognizes and outputs operator names, critical timing information, keywords, and decomposition information about the operators contained in the PSDL source code. The Kodiyak program used in this implementation can be found in Appendix B.

The critical timing information includes maximum execution time (MET), minimum calling period (MCP), minimum response time (MRT), period and the time within which the operator must complete its operation. The keywords include MET, MCP, MRT, PERIOD, WITHIN, LINK, LINEAGE, ATOMIC and END LINEAGE. The keywords are necessary to allow the package FILE_PROCESSOR to know what type of information is to be processed. The keywords MET, MCP, MRT, PERIOD, and WITHIN all identify critical timing information which must

be associated with the various operators that comprise the PSDL source code. The keyword LINK signals that the following information will be precedence relations between the operators. The keywords LINEAGE and END LINEAGE alert FILE_PROCESSOR to information about the decomposition of one operator into other operators. The keyword ATOMIC signals that an operator has not been decomposed further and that its implementation will be in Ada. All other information contained in the original PSDL source code is ignored by the Static Scheduler's attribute grammar processor.

B. THE PROGRAMMING LANGUAGE ADA

Ada is a modern algorithmic language that has the usual control structures, and has the ability to allow the user to define types and subprograms. Ada has considerable expressive power that enables it to cover a wide application domain. It allows for modularity, where data, types, and subprograms can be packaged. It also supports modularity in the physical sense with facilities for separate compilation of program units.

The language was designed specifically for real-time programming, with facilities to model parallel tasks and to handle exceptions. Ada also supports systems programming where precise control over the representation of data and access to system-dependent properties is necessary.[Ref. 13] Ada is a general-purpose computer programming language developed in response to the government wide crisis in software development. It was developed for constructing large computer programs that are to be used in embedded computer systems. Embedded systems are those such as are found in aircraft or missile guidance systems, command and control systems, and computer-controlled radars or weapons. Typically, these systems are constructed by large teams of

programmers, take several years to develop, and have lifetimes spanning several decades, during which time the programs are often upgraded, corrected, and modified.[Ref. 14]

The development of Ada was a direct result of the fact that the Department of Defence (DOD) realized that it was spending too much time, effort, and money developing and maintaining software for embedded computer systems [Ref. 15]. The majority of the costs were not incurred for developing new systems but were incurred in the maintenance of old systems. Hundreds of models of computers and over 450 general-purpose programming languages and dialects were being used for DOD embedded computer systems. Much of the software involved was being written in assembly language to overcome deficiencies in the high-level languages being used and to accomplish functions not amenable to high-level implementation. The lack of a common language and the use of assembly language makes the development of new software difficult and creates even more serious problems for software maintenance.[Ref. 14]

As a result the Department of Defense decided to investigate the feasibility of having a single language for all of its embedded systems and general purpose applications. The following excerpt from the *High Order Language Evaluation Project Final Report* [Ref. 16] summarizes the desired features of the high-level language DOD required.

> The basic concept of a high-level language is to provide the programmer with a set of facilities which are meaningful in the terms of the problem being solved rather than in terms of the machine on which it is being solved. Early research in high-level languages confirmed that programmers were more productive and produced higher quality code when working with a high-level programming language...Recent work in programming methodology has gone on to validate that concept still further. Abstract data types remove still another level of object presentation throughout much of a program. The

> programming language is seen as an extension of the problem-solving process; it is desired that the transformation from a program design (problem solution) to a correctly functioning program be simplified as much as possible. The value of abstraction is to hide the lower level details of implementation from the programmer. The value of modularity is to manage the intellectual complexity of the programming task, to make it feasible to divide a large project into pieces, and to permit parts of a program to be isolated is such a way as to minimize side effects and the problems of connecting modules.[Ref. 16:pp. 3-4,3-5].

Ada was the ultimate result of DOD's desire to have a single language. It does an excellent job of implementing the principles just outlined. It has facilities for allowing the programmer to implement abstract data types together with associated operations on the data types allowing the problem solution to conform to the abstract terms of human thought. Abstraction is taken another level further in that Ada has a facility, called a generic package, that allows a module to be written once and instantiated many times with different types of data. This is one of its best features because it allows programmers to generalize problem solutions.

Modularity is another important positive feature of Ada. It allows program specifications and implementations to be written and compiled in separate modules enabling the programmer to concentrate on one step of the problem solution at a time. Modules also are important for program maintenance, in that the implementation of modules can be modified or replaced without affecting the rest of the program.

Ada also supports information hiding with packages, which are a special type of module that "packages" some concept or data type abstraction. These packages enable the programmer to hide internal objects from the user of the package. In addition, Ada's private and limited private types enable the programmer to hide the details of the construction of a data type from the user. These features reduce the

complexity with which one must deal and enforce a level of security and, therefore, safety into programming.

Ada is a very safe language in which to program. The strong typing that is enforced, in concept by design, and in actuality by the compiler, makes it impossible to accidentally mix operations on the data being manipulated by the program. In addition, Ada provides an exception handling facility which allows the program to handle conditions that may lead to errors or program failure. This is a critical feature in embedded systems, where there can be no programmer interaction.

Four of the Static Scheduler's modules are written in Ada. Ada was chosen for the implementation of the Static Scheduler for two reasons. First, as was discussed earlier, Ada is the Department of Defense's language and is specifically designed for use in embedded systems and contains constructs necessary in these systems. It is envisioned that CAPS will be used primarily to ease the construction of embedded software systems by producing prototypes for these software systems in Ada. Second, the use of Ada in the construction of CAPS supports DOD's efforts to use a single language and demonstrates that Ada can be used as a general purpose high-level language. The Static Scheduler ultimately produces Ada source code which will be compiled with other Ada code from the Translator and Dynamic Scheduler to produce the final executable prototype.

Many of Ada's special constructs are crucial to the successful implementation of the Static Scheduler. The facilities for separate compilation of program units make it possible to modularize the design and encapsulate the abstract data types for the Scheduler. Generic program units make possible the use of several previously written abstract data types such as lists, trees and variable length strings which are

used as data structures within the Scheduler. The Ada task provides a synchronization mechanism between parallel tasks which makes it possible for the Dynamic Scheduler to combine the Static Schedule, produced as one output by the Static Scheduler, with the non-time critical operators that are produced as the other output. Parallel tasks may be implemented on multiple processors or with interleaved execution on a single physical processor with the same result [Ref.13]. Ada also contains pragmas which are instructions to the compiler about how a program is to be constructed. Although the Static Scheduler is written without tasks and pragmas, both language constructs are critical in order for the Execution Support System to function properly. A listing of the Ada source code for the Static Scheduler can be found in Appendix D.

IV. DEVIATIONS FROM THE ORIGINAL DESIGN

In the actual implementation of the Static Scheduler some deviations from the original design outlined by [Ref. 4] and [Ref. 5] were necessary. Different data structures were used than those outlined in the implementation guide for the purposes of efficiency. Provisions for inclusion into the Static Scheduler's Debugger were embedded for exception handling so that the integration of the Execution Support System would require as few changes as possible to the Static Scheduler's source code because the Static Scheduler is not a stand alone system. Some additional modifications to the Scheduler were included that were necessary for practicality, and in order to make it function correctly. During the implementation several difficulties with the design were discovered. These will be discussed in the following chapter.

A. DATA STRUCTURES

Three major data type abstractions are used in the current implementation of the Static Scheduler. They are as follows:

- OPERATORS
- LINKS
- PRECEDENCE.

OPERATORS contains all the operators from the PSDL source code, together with their critical timing and control constraints. LINKS contains all of the operators together with their connecting data streams and maximum execution times. PRECEDENCE specifies an order of execution of all of the operators.

The data structures for these data type abstractions were originally described by Janson [Ref. 5] as a set of files that are arrays of records. However, continual manipulation of files to obtain, rearrange and store data is very cumbersome and is too time consuming and input/output (I/O) intensive. Additionally, the Static Scheduler must find and use the information in the data type, in an order that is not always sequential. This alone makes the use of files impractical. As a result, the record structure of each instance of the data type was kept as originally described, but data structures were selected and implemented for the major data types used by the Static Scheduler. The correspondence between the abstract data types in the design and the data structures used in the implementation is shown in Table 1. These abstract data types are all encapsulated in an Ada package called FILES in order represent the way humans generally think of storage and retrieval of information.

TABLE 1. DATA TYPES AND THEIR CORRESPONDING DATA STRUCTURES

Abstract Data Type	Data Structure
OPERATORS	N_ary Tree
LINKS	Linked List
PRECEDENCE	Linked List

Files were only used for the storage of information that would be used outside the Static Scheduler by some other portion of the Execution Support System. Even these files can eventually be transformed into data structures once the Execution Support System is complete, in order to speed up the translation process of the PSDL prototype to the executable prototype.

1. The N-ary Tree

A single operator within the OPERATOR type was implemented as a record with six fields as originally designed. These fields are shown in Table 2. This information identifies each operator and presents their timing and control constraints. Although implementation began with the data structure for OPERATORS being a linked list, it became obvious that decomposition information about each operator was also necessary for validity checking on the critical timing information, so another data structure was required.

TABLE 2. RECORD FIELDS FOR OPERATORS

FIELDS	CONTENTS
THE_OPERATOR_ID	the name of the operator
THE_MET	maximum execution time for the operator
THE_MRT	maximum response time for the operator
THE_PERIOD	the operator's period
THE_WITHIN	the time within which the operator must finish

The additional information that was required about OPERATORS was the relationship of one operator to other operators within the decomposition of the prototype. Figure 4 shows the relationships between operators in the decomposition process. In particular, it is necessary to know which operator an individual operator is decomposed from, and what operators are decomposed from this operator. For instance, in order to validate the METs of an operator's decomposition, it is necessary to know the name of the operator, its MET, and the names of the operators to which it decomposes and their METs. The PSDL decomposition of operators can be visualized as a tree of operators because any

operator will be decomposed into a finite set of operators, which can be thought of as the operator's children. These children are all on the same level with each other. Therefore, each level of decomposition can be thought of as a level within the tree, so it is natural to represent the data type for OPERATORS as a tree as is shown in Figure 4. The root of the tree is the top level of the decomposition where the design of the system begins. Each successive level within the tree represents a further decomposition of the operators.

In this manner, a tree traversal can obtain the parent operator, that is, the operator from which the operator under consideration was decomposed, as well as the children, or operators to which the operator decomposes. For example, it is apparent from an inspection of Figure 4 that operator C decomposes into operators G and H, so C is the parent of G and H and that G and H are the children of C. With this information it is now possible to validate the METs of C, G and H.

Due to this necessity for composite to component decomposition information, the data structure chosen for the type OPERATORS is a generic n-ary tree. An n-ary tree, as it is implemented here, is a rooted tree in which any node in the tree can have any number of children. This is necessary to accommodate any decomposition of an operator that the system designer using CAPS could construct. Figure 4 is also a representation of an n-ary tree.

The n-ary tree abstract data type was implemented as a generic tree, so that any data type could be stored in the nodes of the tree. In the case of the Static Scheduler, the nodes are of the type OPERATOR. A wide variety of functions and procedures were encapsulated in the n-ary tree abstract data type enabling the user to operate on the tree without knowledge of its internal structure. These operations include, but are not limited to the following:

- IsEmpty -- determines if the tree is empty
- InsertRootNode -- insert a node as the root of the tree
- InsertChild -- insert a child node of the current node
- UpdateNode -- update the information in a node
- FindChild -- find a child of the current node
- FindParent -- find the parent of the current node
- DeleteNode -- delete a node from the tree
- NumChildren -- find the number of children of the current node
- FindRoot -- make the root node the current node
- RetrieveNode -- retrieve the information in the current node.

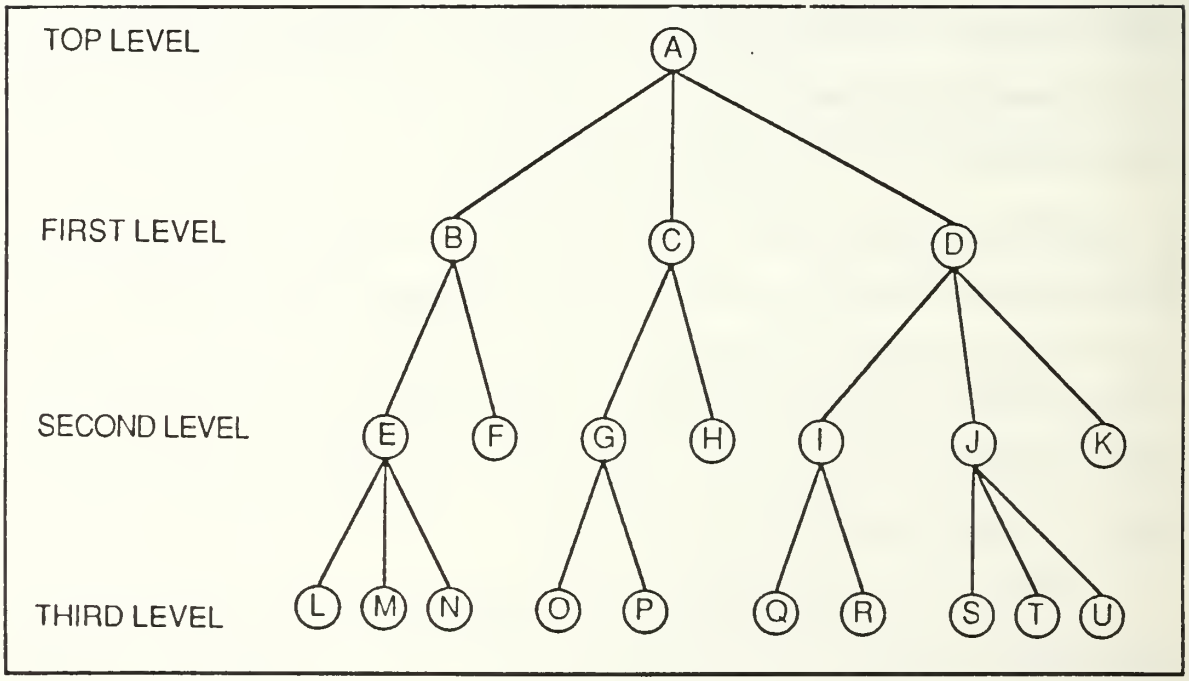


Figure 4. PSDL Operator Decomposition in an N_ary Tree

Operations on the tree are easy to use. However, care must be taken to read and understand the specifications for the data structure because each operation on the data structure affects which node of the tree is the current node. This means

that many of the operations on the tree may affect which node is the current node, that is, the node on which all operations are performed. The following is a simplified version of the structure of the N-ary tree specification.

```
with LISTS, TEXT_IO;
generic
  type NARY_TOKEN is private;
package N_ARY_TREE is
  type NARY_TREE is private;

  function IsEmpty (T : NARY_TREE) return BOOLEAN;
  procedure InsertRootNode (T : in out NARY_TREE;
                            El: in NARY_TOKEN);
                .
                .
                .
end N_ARY_TREE;
```

A complete listing of the specification and implementation of the n-ary tree abstract data type can be found in Appendix C.

2. Linked Lists

A single instance of the type LINKS was implemented as a record with four fields. These four fields are shown in Table 3, and can be used to convey all of the information in a single link statement. Figure 5 shows the relationship between the three representations of a link statement. Link statements and their construction from the graphical representation of an operator's decomposition is more thoroughly discussed in Thorstenson's thesis on the Graphical Editor [Ref. 17].

TABLE 3. RECORD FIELDS FOR LINKS

FIELDS	CONTENTS
THE_DATA_STREAM	the name of the data stream
THE_FIRST_OP_ID	the name of the first operator
THE_LINK_MET	the maximum execution time for the first operator
THE_SECOND_OP_ID	the name of the second operator

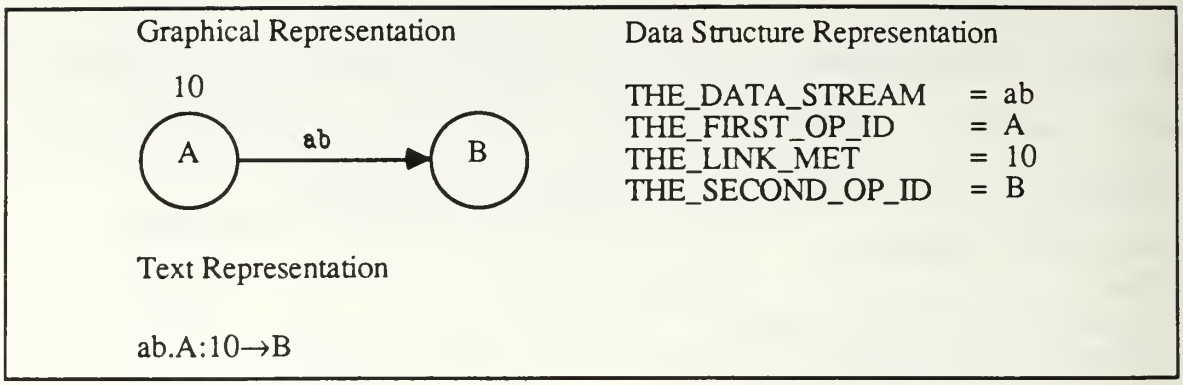


Figure 5. Three Representations of a Single Link Statement

A list data structure was chosen for this data type. A simple unordered list of LINKS is all that is required for the implementation. These links will appear in the PSDL source in the order of their operator's decomposition, which will be from the top level down. A particular ordering of the LINKS is relatively unimportant because it is the information conveyed in THE_DATA_STREAM, THE_FIRST_OP_ID, and THE_SECOND_OP_ID fields which is required by the Static Scheduler.

The third major data type used in the Static Scheduler is the type PRECEDENCE. It is a record which consists of only two fields, THE_LEFT_OP_ID, and THE_RIGHT_OP_ID, which are shown in Table 4. A list data structure was also chosen for the implementation. A linear traversal of the PRECEDENCE list will give the order of execution of operators in the final Static Schedule that is constructed by the Static Scheduler.

The list abstract data type chosen for the implementation of the LINKS list and PRECEDENCE list is a generic linked list package that appears in Britnell's thesis on data structures in Ada [Ref. 18]. This abstract data type was originally to be used for all of the major data structures within the Static Scheduler until it was

determined that a list would be insufficient to convey all of the required information about OPERATORS.

TABLE 4. RECORD FIELDS OF PRECEDENCE

FIELDS	CONTENTS
THE_LEFT_OP_ID	the name of the operator appearing as THE_FIRST_OP_ID in a LINK statement
THE_RIGHT_OP_ID	the name of the operator appearing as THE_SECOND_OP_ID in a LINK statement

The linked list package used in this implementation contains several functions and procedures. These include:

- Clear -- Makes the list empty
- Full -- Determines if the list is full
- Empty -- Determines if the list is empty
- Insert -- Inserts a node in the list
- Delete -- Deletes a node from the list
- Length -- Determines the length of the list
- Find_Item -- Finds an item in the list
- Find_Pos -- Finds the position of an item in the list.

The only difficulty encountered with this package is that the deletion procedure is designed to delete an item from a list based on its position within the list. This makes it virtually impossible to delete more than one item at a time or to delete within a loop because a deletion alters the position of all the subsequent items in the list. As a result, it is much easier to build a new data structure, simply leaving out the items that would be deleted from the list, than it is to actually delete them from the original data structure. The following is a simplified specification of the linked list package.

```
generic
  type Item is private;
package Generic_List is
  type List is private;

  procedure Clear (L : in out List);
  function Full (L : in List);
        .
        .
        .
end Generic_List;
```

The complete specification and implementation of this linked list can be found in Appendix C.

3. Variable Length Strings

The Ada language has a predefined "string" type, but this could not be used as the base type for the operator and data stream fields within the OPERATORS, LINKS, and PRECEDENCE types because a string is a fixed length. Since these fields are necessarily of a variable length to accommodate the Ada identifiers that would be assigned to them, a variable length string abstract data type was necessary. A generic variable length string package from a public domain library was chosen for the implementation. It has functions to convert a standard Ada string to a variable length string, functions for comparison, and procedures for input and output. These were the main functions necessary for the Static Scheduler, though there are many others in the package. The following is a simplified version of the specification showing the functions used.

```
with TEXT_IO; use TEXT_IO;
generic
  LAST : NATURAL:
package VSTRINGS is
  subtype STRINDEX is NATURAL:
  FIRST : constant STRINGDEX := STRINDEX'FIRST + 1;
  type VSTRING is private;
```

```
   NUL : constant VSTRING;

   function ">=" (LEFT: VSTRING;
                  RIGHT: VSTRING) return BOOLEAN;
   procedure PUT_LINE (ITEM : in VSTRING);
   procedure GET_LINE (ITEM : in VSTRING);
   function VSTR (FROM: STRING) return VSTRING;
                  .
                  .
                  .
end VSTRINGS;
```

Use of the package is very simple, and a complete listing of the specification and implementation for the variable length strings abstract data type can be found in Appendix C.

B. EXCEPTION HANDLING

The Static Scheduler is designed to build a static schedule from the PSDL source file unless conditions are found which would make the construction of the schedule unfeasible. If none of these conditions are found, it will construct a schedule for the operators that make up the prototype. If any condition is found that makes construction of a valid schedule impossible, an exception is raised to notify the designer that a schedule is unfeasible with the information provided. The type of exception raised identifies what the problem is.

Unfortunately, the exception handling facilities of Ada are insufficient for the Static Scheduler within CAPS, because an exception can tell the designer what the problem is but not where it is. In a large system, this is clearly not sufficient. As a result, another method had to be found to tell the designer where, within the system being designed, the problem occurs. The implementation of the Static Scheduler currently uses the standard Ada exceptions, but also includes the code necessary once the Scheduler is integrated into the Debugger and the Execution Support

System. Since the Debugger needs to tell the designer of the prototype where the trouble that caused the Static Scheduler to fail occurred, it is necessary for the Debugger to have the name of the operator where the problem was detected. This will allow the designer to examine information about the operator, and then, based on the type of exception raised, determine where in the operator or in its decomposition the problem occurred so that it can be corrected.

Since the Static Scheduler will be embedded within the Debugger of the Execution Support System, the operator's name, and the exception type must be passed to the Debugger and then control turned over to the Debugger. As envisioned in the design of the Debugger [Ref. 9] the Static Scheduler will be a task within the Debugger. When the Static Scheduler discovers an exception the following will occur. A variable, named Exception_Operator, must be set by the Static Scheduler and a procedure call to the Static Scheduler Debugger made to transfer control to the Debugger. This will allow the Debugger to correctly handle the exception and give the designer the name of the operator that caused the exception. This is done in the Static Scheduler by having a global variable named "Exception_Operator" set by the Scheduler whenever an exception condition is discovered. Then the exception is raised. An example of this would be as follows for the package FILE_PROCESSOR in the case where the a critical operator (one with timing constraints) lacks an MET.

```
Exception_Operator := CurrentOp.THE_OPERATOR_ID;
raise CRIT_OP_LACKS_MET;
```

Once the exception has been raised an exception handler is looked for in successive outer levels of control if not found at the bottom of the code in which the exception is raised. In the case of the Static Scheduler, the exception handler is

located in the driver program, so control is returned to this piece of code. The exception handling will already have set the global variable Exception_Operator, and the actual exception handler appears as follows.

```
when FILE_PROCESSOR.CRIT_OP_LACKS_MET =>
  SS_Debug.CRIT_OP_LACKS_MET;
```

In effect this is a procedure call to the Static Scheduler Debugger. The Debugger notifies the designer of the exception and the operator which caused the exception. Once that notification is complete, control is returned to the Static Scheduler which then ends execution without producing a Schedule.

C. PACKAGE IMPLEMENTATION

Of the five packages in the architectural design of the Static Scheduler, the first three have been implemented, as well as the package FILES that encapsulates the abstract data types used. These include

- PSDL_READER,
- FILE_PROCESSOR, and
- TOPOLOGICAL_SORTER.

The Kodiyak source code for the package PSDL_READER can be found in Appendix B. The Ada source code for the packages FILES, FILE_PROCESSOR, and TOPOLOGICAL_SORTER can be found in Appendix D while the source code for their data structures can be found in Appendix C. The differences between the design and implementation are outlined in this section.

1. The FILES Package

The package FILES remains essentially as originally specified in Janson's implementation guide [Ref. 5] for the Static Scheduler. Only three small deviations were necessary for this package.

The first change was to import and instantiate the variable length string discussed earlier in this chapter because it is an essential data structure for the implementation. This enabled the implementation to make the operator names and data stream names of variable length, up to a maximum of 80 characters. The number of characters allowed was chosen rather arbitrarily and can be easily changed if it proves inadequate. However, it seems that an Ada identifier of more than 80 characters would not normally be necessary.

The second change was to make the values allowed for the critical timing information within the data types natural numbers rather than strings to correspond with PSDL and make comparison of values within these fields simpler.

The last deviation was to instantiate packages for each of the data types given. As was discussed earlier, this included an n-ary tree for OPERATORS, and linked lists for the atomic OPERATORS, LINKS, and PRECEDENCE. With this encapsulation of all of the major data structures within the Static Scheduler implementation could proceed on the rest of the packages.

2. The PSDL_READER Package

The first major deviation from the architectural design outlined in the implementation guide by Janson [Ref. 5] is in the package PSDL_READER. Two things are to be done in this package. The first is to invoke the attribute grammar processor, and the second is to remove some extraneous information from the attribute grammar processor's output file.

The attribute grammar processor is the PSDL translator discussed earlier in Chapter III of this thesis. It is an executable program that accepts a PSDL source program as input and produces a file containing the information required by the Static Scheduler. Since this is an executable file, and Ada does not have facilities

within its compiler to import executable code, the attribute grammar processor must be invoked by the sequence control function of the User Interface of CAPS. The attribute grammar processor cannot actually be embedded in the Ada source code that comprises the rest of the Static Scheduler so it is actually a separate process that must be done before the rest of the Static Scheduler can be invoked.

Within the attribute grammar processor described in the implementation guide [Ref. 5] some information is collected and labeled JUNK for further processing. This implementation has the attribute grammar processor simply ignore this information since it is not needed by the scheduler, therefore it is not collected. The second thing the original PSDL_READER does is to discard the information that was labeled JUNK. Since this information is no longer collected, the procedure READ_THE_FILE, within PSDL_READER, which would have removed JUNK from the input file, has been eliminated. The entire package PSDL_READER has become simply the attribute grammar processor.

3. The FILE_PROCESSOR Package

The second package, FILE_PROCESSOR, remains as originally outlined however the two procedures within it, SEPARATE_DATA and VALIDATE_DATA were modified for various reasons. In addition to the exceptions identified in the original design several new exceptions were identified and implemented in the Static Scheduler in the procedure VALIDATE_DATA within the FILE_PROCESSOR package. These include

- CRIT_OP_LACKS_MET
- MET_REQUIRED
- MET_GT_PARENT
- MET_SUM_GT_PARENT
- MET_NOT_LESS_THAN_PERIOD.

In fact, of the two exceptions outlined for this package in Janson's thesis [Ref. 5], only one exception, MET_NOT_LESS_THAN_MRT, remains as originally discussed.

In addition recursive procedures were implemented in order to be able to traverse the n_ary tree data structure of OPERATORS. These include the two procedures

- FindOperator -- locates an operator in the tree
- TraverseOps -- enables the operator names to be printed.

In the implementation guide, the non-time critical operators were to be separated from the time critical operators in SEPARATE_DATA. This assumes that time critical operators always have an MET and non-time critical operators never have any timing constraints. It was decided that it would be better to validate this before separating the time critical operators from the non-time critical operators, therefore this function was moved to the procedure VALIDATE_DATA. Other than this and the recursive procedures to accommodate the OPERATORS tree structured data, SEPARATE_DATA was implemented much as originally intended.

Recursion was necessary to operations on the n_ary tree because the tree can be of any size and shape. This means that there may be any number of nodes at any level in the tree, and each node must be traversed in order to find a specific operator within the tree in advance. Because it is impossible to know the depth of the tree, or the number of nodes within the tree, only recursion can guarantee that all nodes are visited. Therefore, the procedure FindOperator was implemented as a recursive tree traversal, to ensure that an operator would be found if it is in the tree.

The procedure VALIDATE_DATA is perhaps the most important individual procedure within the Static Scheduler. If the critical timing information given by the designer of the system being prototyped is not valid, there is no possibility that the prototype will be a reasonable indicator that the system under development is feasible. As a result, four validity checks were added to the implementation, one of which was discussed in the design by O'Hern [Ref. 4], but was not covered in the implementation guide. The other three were added because they seemed reasonable and necessary to the proper functioning of the Static Scheduler. Finally, three recursive procedures were added.

- CheckTiming -- ensures critical operators have an MET
- StoreOps -- stores critical operators in a list
- SortAtomics -- sorts atomic operators into a time-critical atomic operator list and the NON_CRITS file.

In addition, several other validity checks deemed necessary will be discussed in Chapter V.

One of the validity checks that was discussed in the design states that the sum of the METs of all of a composite operator's children must be less than or equal to the composite (parent) operator's MET [Ref. 4]. This check was implemented, and an exception called MET_SUM_GT_PARENT was added. This assumes a strictly linear decomposition of the composite operator. This is an invalid assumption for a single processor, since it is more likely that the decomposition of any operator will be a network directed graph rather than a simple linear directed graph. This assumption would not necessarily be invalid for a multiprocessor system, but this implementation is supported by only a single processor. This problem will be discussed further in Chapter V.

The check just discussed immediately suggested another validity check. If a composite operator has an MET, then all of its components, and their decompositions, must have an MET. As a result, this validity check and the exception MET_REQUIRED was added to the implementation. In addition, each of these METs must be less than the composite's MET. If any individual operator's MET in a decomposition is greater than the composite's MET, the decomposition is not valid. Therefore, a check to determine if this is the case and the exception MET_GT_PARENT were implemented.

These validity checks were implemented by using the recursive procedure StoreOps to traverse the n_ary tree and store time-critical operators in a linked list called Ops_with_MET. Then further checks could be performed by a simple linear traversal of the new data structure Ops_with_MET.

The final validity check added to the implementation tests to see if there is an operator without an MET that has other critical timing information such as a MCP, MRT, PERIOD, or FINISH_WITHIN. If this is the case, then the exception CRIT_OP_LACKS_MET is raised. This validity check was added even though the design assumes that a critical operator will always have an MET. It will prevent the designer from unintentionally losing critical timing information caused by the error of leaving out an MET in the specification of an operator, which would occur during separation of the critical from non-critical operators. This validity check was implemented using the recursive procedure CheckTiming.

Once this last validity check is complete it is possible to separate the time critical operators from the non-time critical operators. Another deviation from the design and implementation guide occurs during this process. In the decomposition process from the top level operator to the lowest level operators in the design two

separate types of operators appear. One is the composite operator which can be further decomposed into other operators on the next lower level. The other type of operator is an atomic operator, which cannot be decomposed further and must have an Ada implementation.[Ref. 6] This implementation will either have been retrieved by the software base which is discussed in Galik's thesis [Ref. 3] or directly written for the prototype. These atomic operators will appear in the tree of OPERATORS as the leaves of the tree as is shown in Figure 6.

Only the atomic operators will be scheduled by the Static Scheduler or have translations given by the Translator. The translation of PSDL is more thoroughly discussed in [Ref. 8] and [Ref. 12]. Composite operators need not be scheduled, because their decompositions specify the entire operation of the composite operator. Therefore, execution of all of the atomic operators along the frontier of the tree of OPERATORS will result in the execution of the prototype. As a result, the atomic operators must be separated from the composite operators, for further processing within the Static Scheduler.

This deviation from the design was implemented by performing a recursive traversal of the OPERATORS tree and sending the non-time critical atomic operators to the NON_CRITS file and, at the same time, building a new data structure of the time critical atomic operators called ATOMIC_OPS. This is done by the procedure SortAtomics. ATOMIC_OPS is a linked list of the type OPERATORS, which is then used for all further processing of operators within the Static Scheduler.

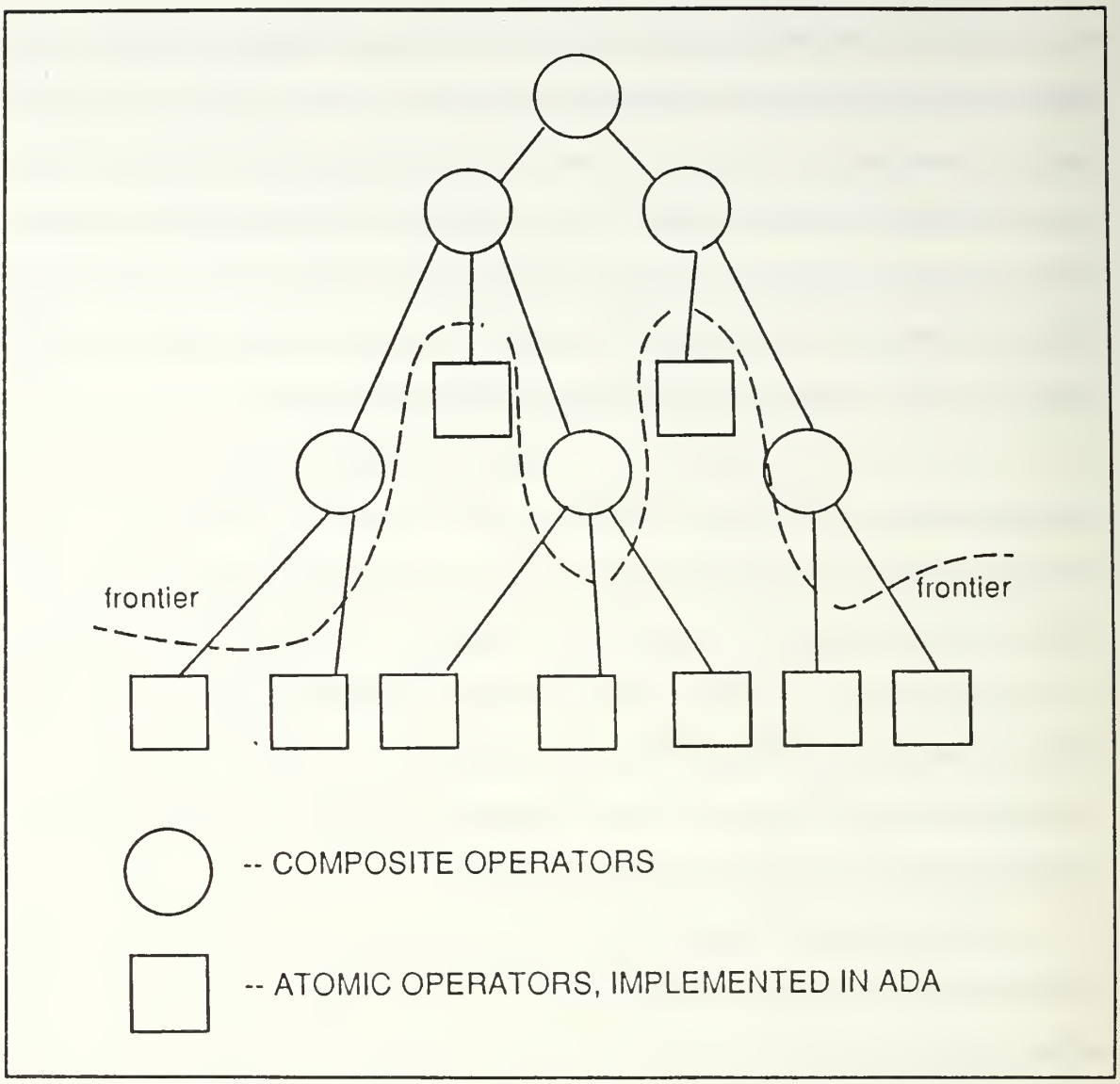


Figure 6. Composite and Atomic Operators in the Operator Tree Data Structure

4. The TOPOLOGICAL_SORTER Package

The TOPOLOGICAL_SORTER implementation contains only the two procedures given by Janson [Ref. 5]. The logic in CREATE_LISTS, however, follows the description given for finding the first operator in [Ref. 4] as it is more accurate. In this procedure the first operator which must be found in order to

create the PRECEDENCE list of the Static Scheduler can be identified by examining the list of LINKS. Operators that must execute first are those that have only external inputs, or those that have no external inputs, and appear only as the left hand operator in a link statement.

Figure 7 shows a typical graph decomposition and its associated link statements as they would be generated by the Graphical Editor implemented by Thorstenson [Ref. 17]. METs for the operators are not shown because they are not relevant to this part of the Scheduler. It can easily be seen from the graph in Figure 7 that either operator A or operator C may execute first since they are not related to one another and do not depend on any other operator's execution. Operator A has only external inputs and operator C has no inputs and appears only on the left hand side of the arrow in the link statement. It also can be seen that B cannot execute first even though it has an external input because it also requires input from operator A. This information is easy to obtain from the graph and can also be found using the link statements.

Using this logic, the operators that could execute first are identified. Next, the list of LINKS is searched to find link statements having these operators on the left hand side. When found, they are inserted, together with their associated right hand operator, into the PRECEDENCE list. Another deviation from the guide is implemented when a check is done to ensure that this precedence statement is not already in the list, as duplicates would be redundant. As can be seen in Figure 7, the link statements

ab1.A --> B and ab2.A --> B,

which have different data streams would cause the operators A and B to be inserted in the PRECEDENCE list twice.

The only two deviations from the implementation guide in the procedure SORT_REMAINING_OPERATORS are that the check to ensure that a precedence statement has not already been inserted was implemented here and the procedure was made recursive. SORT_REMAINING_OPERATORS must be recursive because it continually adds new precedence relations to the precedence list. In order to continue to check these new precedence relations for additional operators to add to the precedence list, it must be recursive. The stopping condition occurs when no new relations have been added to the list.

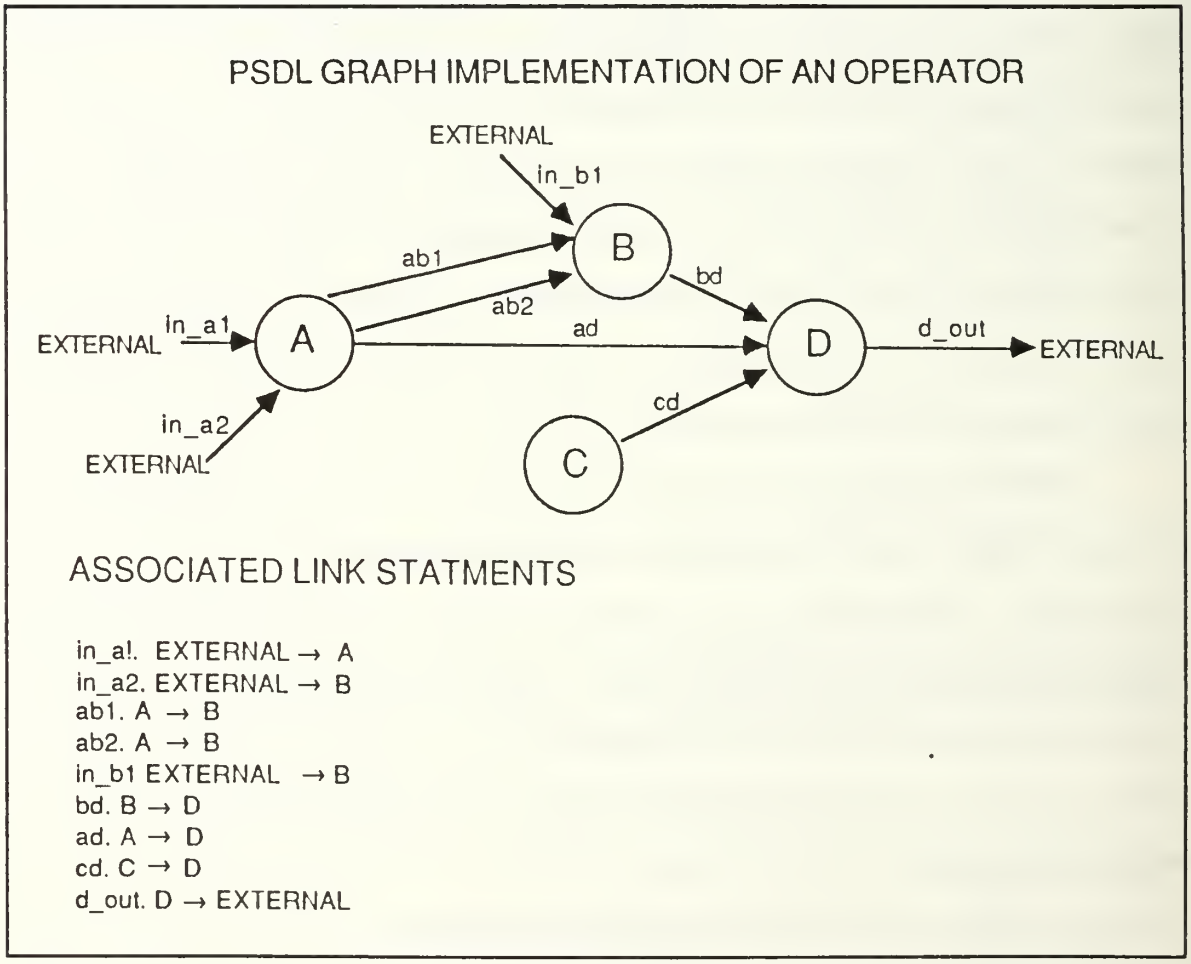


Figure 7. A PSDL Graph and Associated Link Statements

V. DIFFICULTIES WITH THE DESIGN

During the implementation of the FILE_PROCESSOR and TOPOLOGICAL_SORTER packages of the Static Scheduler, six general problems with the design were discovered. Two of these problems deal with validation of the critical timing information beyond that which was discussed in [Ref. 5] and [Ref. 4] and in this thesis. The other four problems deal with more serious issues whose solutions and implementations are critical to the correct functioning of the Static Scheduler.

A. OPERATOR'S AND LINK STATEMENT'S METS

The critical timing constraint, the MET, can appear in two different places in PSDL, in the specification of an operator [Ref. 1] and in link statements from the graph implementation of an operator [Ref. 4]. The MET from these two different sources needs to be compared to see if they are the same. This could be done in either the PSDL Editor for CAPS or in the Static Scheduler itself. If it is done in the PSDL Editor and it can be guaranteed that there are no conflicts with the MET of an operator, it would not need to be done in the Scheduler. However, in order to thoroughly validate the data that the Scheduler is given, this problem could be resolved by either of the following two methods.

If the MET for a particular operator in the OPERATORS data structure is different from the MET for that operator in the LINKS data structure, an exception, perhaps called DIFFERENT_METS, should be raised so the designer can verify which MET is the one desired for that operator. If there is a MET for an operator within the LINKS, but not in the OPERATORS structure, then the MET

in LINKS for that operator should be assigned to the MET within the OPERATORS data structure. The reverse of this operation on the data need not be performed because the MET in the LINKS data structure is never used by the Static Scheduler.

An alternative solution would be to eliminate the MET field from the LINKS structure altogether since only the MET within OPERATORS is actually used by the Scheduler. This would remove the need for this validity check and exception. This solution would also involve altering the attribute grammar processor to ignore the MET in a link statement, and the procedure SEPARATE_DATA in the FILE_PROCESSOR package, so that it would not try to collect this information.

B. COMPOSITE AND ATOMIC LINK STATEMENTS

A much more serious problem for the Static Scheduler is that the current implementation does not completely account for the two different kinds of operators, composite and atomic. As was discussed earlier in this thesis, only atomic operators will be scheduled by the Scheduler. Therefore, all composite operators must be eliminated from the data structures within the FILE_PROCESSOR package before further processing. The problem of removing the composite operators from the OPERATORS data structure was accomplished in this implementation by building a new data structure containing only the atomic operators and has been previously discussed.

However, this has not yet been done for the LINKS data structure. If every link statement contained only atomic or composite operators, it would be a simple problem to remove all of the composite operators by building a new data structure without the composites. This could be done by comparing the operator names within the OPERATORS tree to the operator names within the LINKS list. If both

of the operators in the link statement are atomic, the link statement would be inserted in an atomic LINKS list. If both of the operators are composite, it would not be inserted because composite link statements will ultimately be decomposed in to either composite to atomic link statements or atomic to atomic link statements.

The problem occurs with link statements that have both an atomic operator and a composite operator in them. These link statements cannot be inserted in the list of atomic links and they cannot be discarded, as they contain information that will be vital in building the precedence list. Figure 8 illustrates the relationships between composite and atomic operators and shows the link statements that are generated in the decomposition process. The asterisks that appear in the graph and beside the link statements indicate where combinations of composite and atomic operators occur in the link statements in this example. At the first level of decomposition, operator B is a composite operator, while operator C is an atomic operator, as is indicated by the frontier line.

In order to eliminate the composite operators all those link statements at the top level can be removed without any loss of information to the system. At the second level, the link statement

in_a1.EXTERNAL -> B

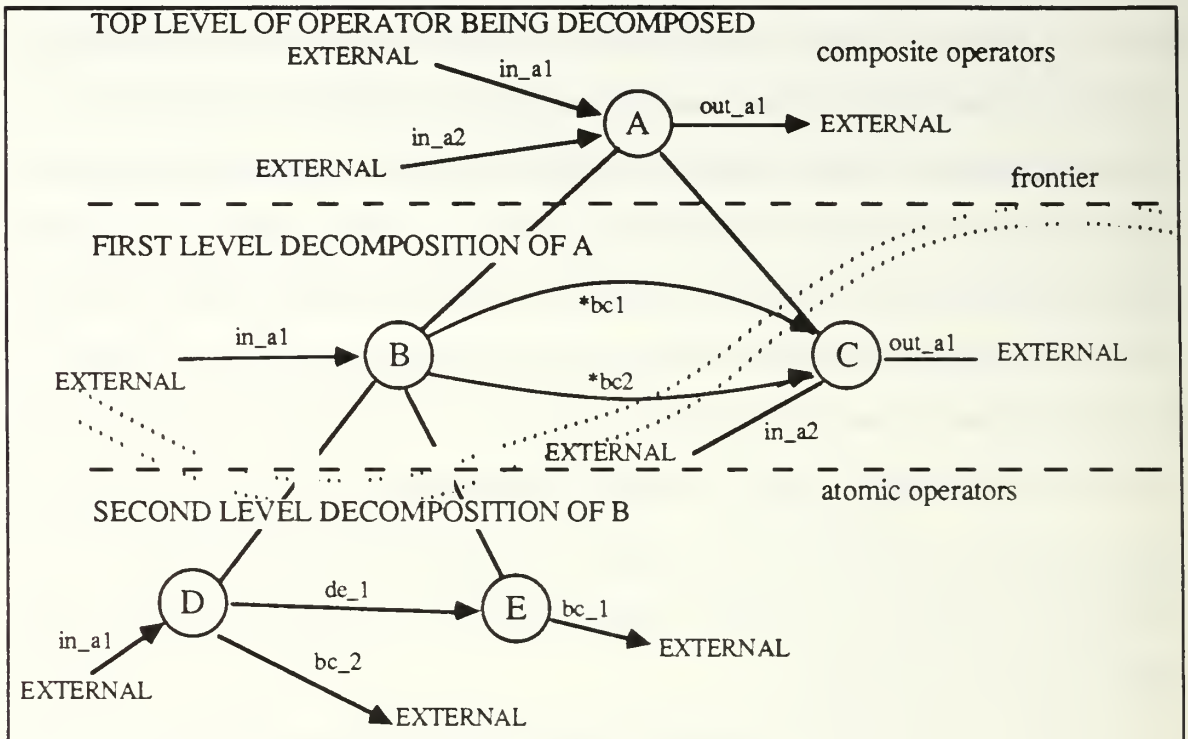
can also be discarded because it contains only a composite operator. The link statement

out_a1.C -> EXTERNAL

contains only an atomic operator so it is inserted in the atomic LINKS list. However, the link statements

bc_1.B -> C and bc_2.B -> C

contain both an atomic and composite operator. In order to reduce these link statements to ones containing only atomic link statements the composite operators must be replaced with atomic operators within these link statements.



TOP LEVEL LINK STATEMENTS
in_a1.EXTERNAL→ A
in_a2.EXTERNAL→ B
out.a1_A →EXTERNAL

FIRST LEVEL DECOMPOSITION LINK STATEMENTS
in_a1.EXTERNAL→ B
in_a2.EXTERNAL→ C
*bc1.B→ C
*bc2.B→ C
out_a1.C→ EXTERNAL

SECOND LEVEL DECOMPOSITION LINK STATEMENTS
in_a1.EXTERNAL→ D
de1.D→ E
bc2.D →EXTERNAL
bc1.E→ EXTERNAL

Figure 8. Composite to Atomic Relationships in a Decomposition

This can be done by searching the composite operator's decomposition for the case in which the data streams, bc_1 and bc_2, are in a link statement where one operator is atomic and the other operator is the word "EXTERNAL". In this case the relevant link statements are found in the second level decomposition of Figure 8. They are

bc_1.D -> EXTERNAL and bc_2.E -> EXTERNAL.

Once located, the link statements

bc_1.B -> C and bc_1.E -> EXTERNAL

must be combined and replaced with the link statement

bc_1.E -> C,

and the link statements

bc_2.B -> C and bc_2.D -> EXTERNAL

must be combined and replaced with the link statement

bc_2.D --> C.

These new link statements are then inserted in the list of atomic LINKS. Once this has been done the only link statements remaining are

in_a1. EXTERNAL -> D and de_1.D -> E

which both contain only atomic operators so they are inserted in the atomic LINKS list. After this process has been completed, the link statements in the atomic LINKS list would be as follows.

in_a1.EXTERNAL -> D
in_a2.EXTERNAL -> C
out_a1.C -> EXTERNAL
bc_1.E -> C
bc_2.D -> C
de_1.D -> E

With these link statements all the requisite information for proper execution of the prototype can be determined. The external inputs and outputs have been preserved for the system and all link statements are expressed only in atomic operators. These atomic operators ultimately specify the execution of the prototype.

In this simple example finding the appropriate atomic link statements and replacing them and their corresponding composite link statement with the correct new atomic link statements was unambiguous. This assumes that data stream names are passed from level to level in the decomposition process when appropriate. If the same data stream appears in different levels of decomposition with different names, this method will not work. In a more complicated example, where the data stream appears several times in successive levels of decomposition, the matching and replacement process may involve replacement by more than one link statement.

C. NON-TIME CRITICAL OPERATORS AND PRECEDENCE

Having decomposed all link statements to the level where they contain only atomic operators, as just discussed, the next function the Static Scheduler needs to perform is to sort the link statements into a PRECEDENCE list which will specify the order in which the operators must be scheduled for execution within the prototype. Another problem in the design occurs at this point.

There is a strong possibility that some of the operators in the list of atomic LINKS will be non-critical operators. Janson's implementation guide states that the precedence list will contain only time critical operators, [Ref. 5] but this will never be the case. Non-time critical operators cannot be removed from the PRECEDENCE list as they can be from the operator list because it would cause

discontinuities in the PRECEDENCE list causing the Static Scheduler to fail by being unable to find the correct successive operators for scheduling.

One might think that this problem could be solved by replacing the link statements containing the operator being removed with new link statements connecting its input to its output as shown in Figure 9. In this example the indirect connection between operators A and C is through the two link statements

ab.A:10 -> B and bc.B -> C

which is replaced by a direct connection through the link statement

ac.A:10 -> C.

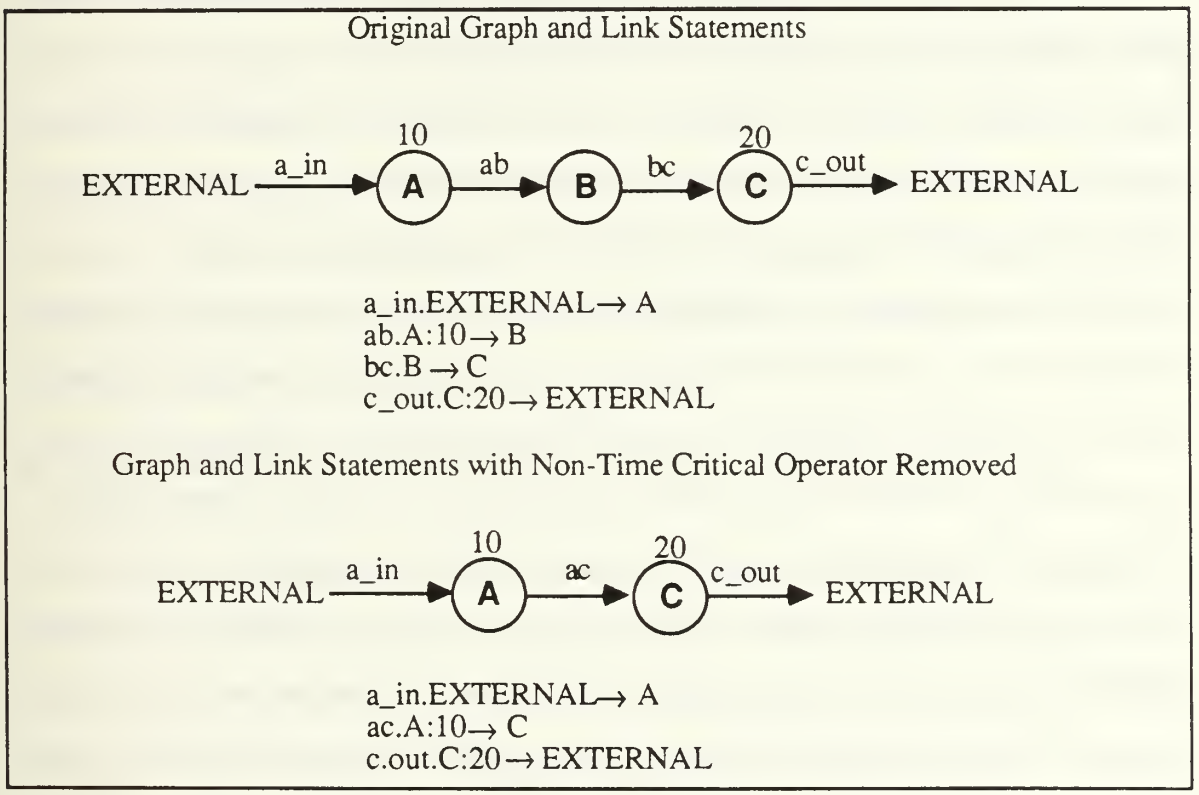


Figure 9. Original Graph and Link Statements

The problem with this solution is that the original graph and link statements indicated that operator B had to execute after operator A, but before operator C.

By removing operator B and sending it to the non-time critical file, NON_CRITS, there is no guarantee when or where the Dynamic Scheduler will execute operator B. Therefore this is not a viable solution. Even in a well formed design, if a time-critical operator reads the output of a non time-critical operator, the data stream must be sampled rather than dataflow, and must have a declared value. The data trigger of B will make sure that B is executed after A, but there is still no guarantee that B will be executed before C. If the stream bc does not have an initial value, then C will try to read an empty stream and cause an exception indicating the design error.

On the other hand, if the non-critical operators have already been sent to the NON_CRITS file, then the Dynamic Scheduler will attempt to schedule them in the empty time slots allowed by the final Static Schedule. In addition, because of precedence relations, which are built from the link statements, the Static Scheduler will attempt to schedule these same atomic operators in the Static Schedule.

There are two problems with this. The first is that non- time critical operators that appear in the PRECEDENCE list will not have any of the timing information required by the Static Scheduler to build the Static Schedule. The design requires that sporadic operators (those without a period) be converted to their periodic equivalents. The algorithm to do this requires sporadic operators to have values for their MET, MCP and MRT. Periodic operators will have at least an MET and PERIOD [Ref. 5]. The second problem is that these operators are scheduled twice, once by the Static Scheduler, and once by the Dynamic Scheduler which will not be the prototype intended by the designer.

This is a very serious problem, which cannot be easily resolved. If the operators within the list of link statements are compared with the list of atomic

operators, and only those that do not appear in link statements are sent to the NON_CRITS file one of the difficulties still occurs. There will be non-time critical operators that the Static Scheduler is trying to schedule that do not have the necessary timing information required to build a schedule.

It appears from the simple examples in Figures 2, 7, and 8 that all atomic operators will appear in link statements and therefore will have precedence relations which must be accounted for. The only case in which operators would not be in the PRECEDENCE list is if there is a discontinuity in the graph decompositions of an operator. This does not make sense as those operators would either not consume data streams, not produce data streams, or both.

If it is the case that all operators have precedence, this problem may be unsolvable within the framework of the current design. The only solution would be to require all operators to have timing constraints, and be scheduled by the Static Scheduler. This would cause the NON_CRITS file to disappear so that the only task for the Dynamic Scheduler would be to combine the Static Schedule with the PSDL Translation and execute the resultant prototype.

D. DECOMPOSITION INTO LINEAR AND NETWORK-LIKE GRAPHS

Another critical problem with the current design and implementation is that the procedure VALIDATE_DATA assumes a linear decomposition of a composite operator into its components. This will be the exception rather than the rule in a decomposition. It is more likely that a decomposition will be a network of operators. Figure 10 illustrates the difference in complexity between a linear decomposition and a network decomposition. These decompositions are shown as they would appear in the Graphical Editor [Ref. 17].

In the linear decomposition it is a simple matter to add the component operators' METs to obtain the sum of 35 in this example. This indicates that this is a valid decomposition of operator A because 35 is less than 38. It is not so easy to do in the network decomposition because every path from each input data stream to each output data stream must be traced and the operators' METs along each path summed. Only if the path with the longest sum of METs is less than the composite operator's MET, is the decomposition valid.

The Static Scheduler does not have the graph representation of a decomposition as depicted in Figure 10 with which to trace all of the paths. Before composite link statements and composite operators are removed, it has only two data structures to give it this information. It has a list of link statements and the tree of OPERATORS. Each operator within the tree also contains the MET information it must use for this validity check.

From this information it must be able to construct all possible paths from each input to the outputs. This is graphically represented in Figure 11 which shows all paths from the two input data streams of the network in Figure 10, represented in a tree form. This is possible because all decompositions will be directed graphs. An operator may be reached only if it has a data stream to it from another operator. The number of trees which must be built for the path search in this visual representation equals the number of inputs in the decomposition. Then each path in each tree must be traversed summing the METs of the operators along the way.

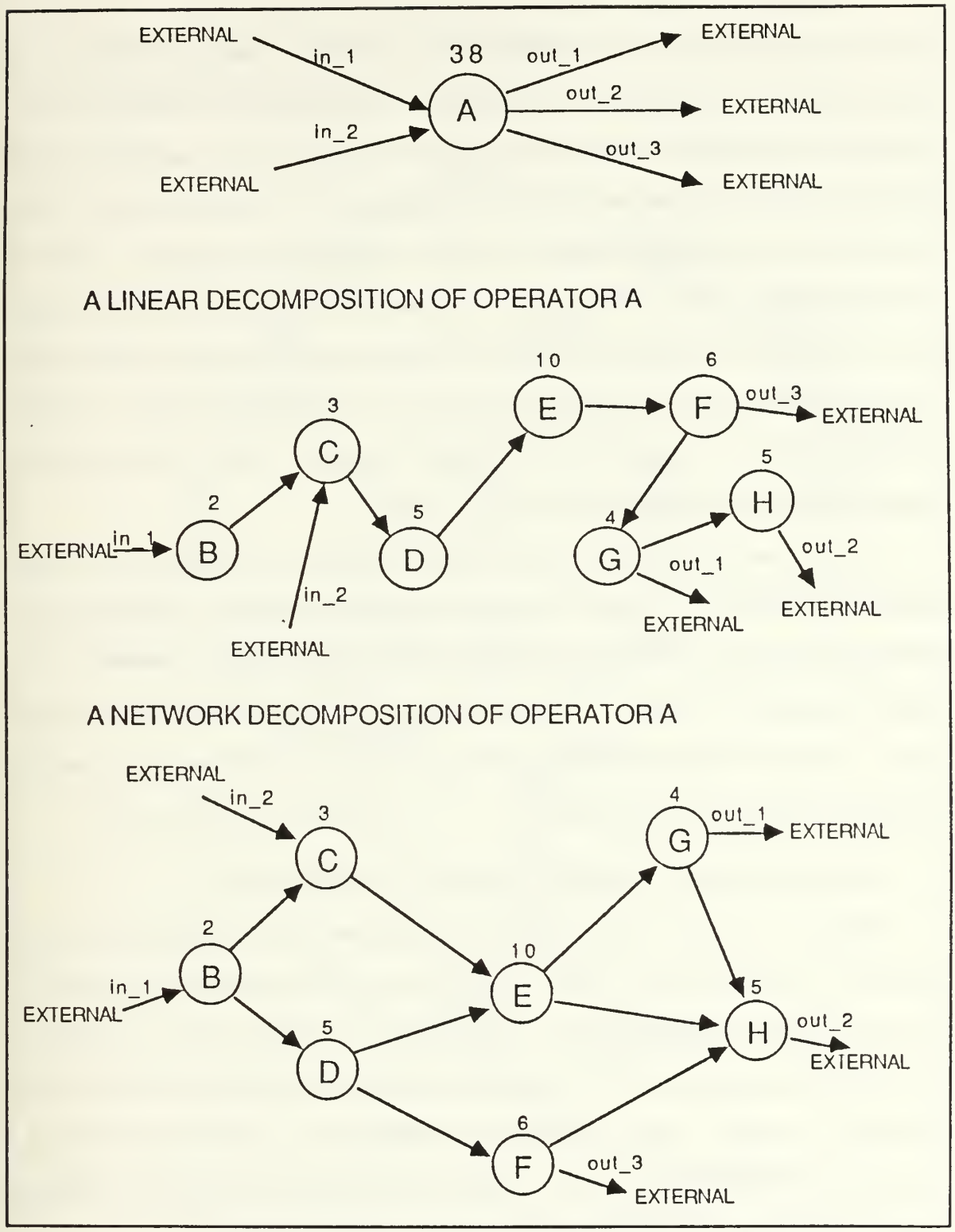


Figure 10. Linear Versus Network Decompositions

This visual method could be used by the Static Scheduler to validate the METs in a decomposition. The Scheduler would have to build each tree using the link statements to determine which operators can be reached from a particular input data stream. Once the first level was built each successive level would have to be constructed using the link statement with their connecting data streams. After a tree is completed, which is recognized when all of the leaves of the tree are the word "EXTERNAL", repetitive traversals of the tree along each path summing the values of the METs and only saving the largest value would be performed. Once all of the input trees have been traversed, the MET sum must be less than the parent operator's MET for the decomposition to be valid.

As can be seen from this example, there is redundant information as the tree starting at operator C is a subtree of the tree starting at operator B. Although this method is fine for visual representation of the problem, as a solution it is cumbersome and will become unwieldy with large decompositions having many inputs. Research needs to be done to find a simple and efficient graph search algorithm to deal with this problem.

E. VALIDATION OF OTHER TIMING CONSTRAINTS

Currently, very little validation of the timing constraints is being done on the operators. Once a solution is found to the network MET problem just discussed, the same method should be used to further validate the other timing constraints. Since composite operators are not scheduled, and are removed prior to the actual scheduling, the MCP, MRT, and PERIOD values of the composites should be compared with their components to ensure that timing information the designer intends to be passed down to component operators does not get lost.

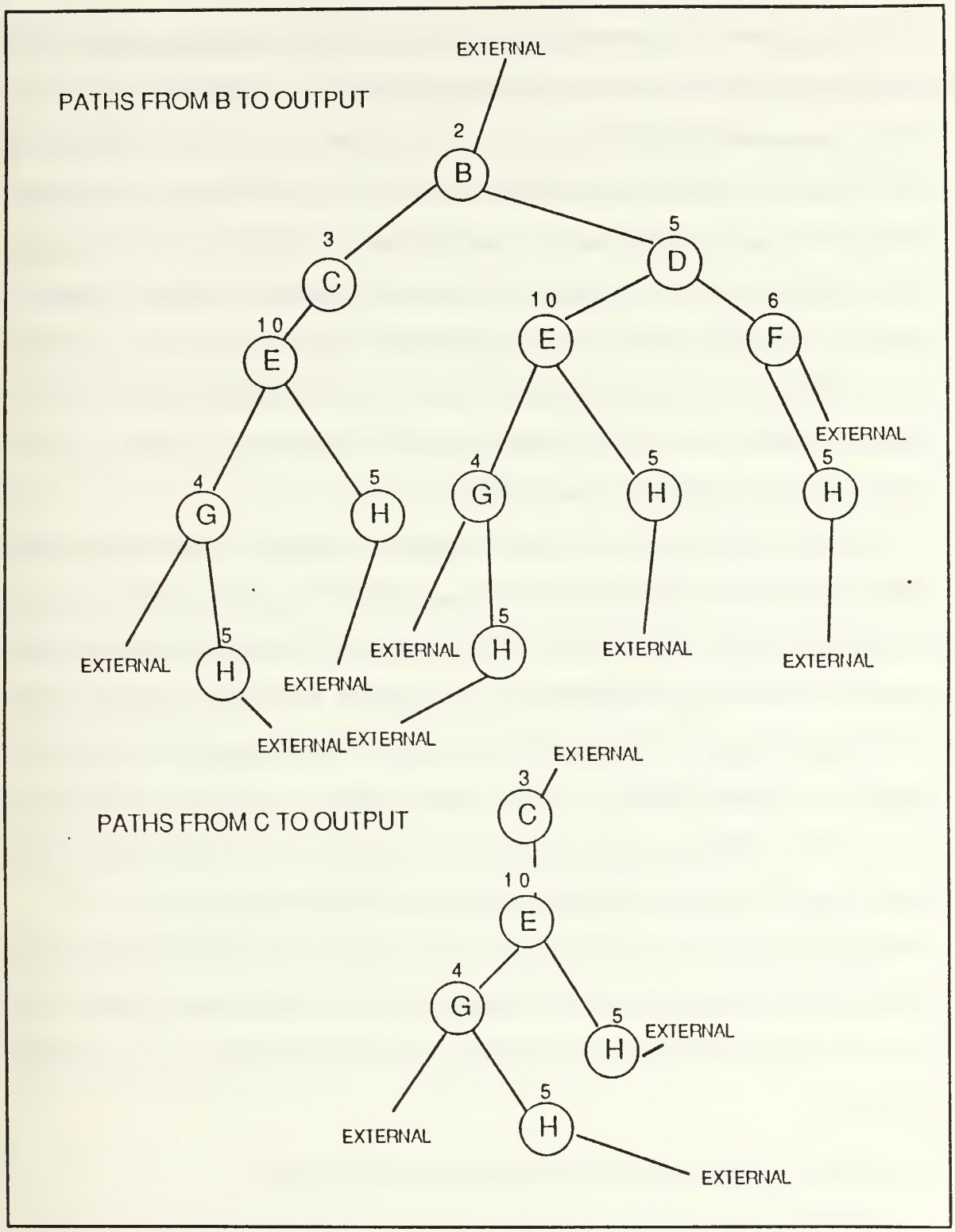


Figure 11. Path Construction from a Network

Validating the MRT for a decomposition would be exactly like that done for the MET. The MRT for the subnetwork of an operator must be less than or equal to the composite operator's MRT [Ref. 1]. This validity check could be done at the same time as the MET validity check by simply using two different variables in which to store the MET sum and MRT sum. If the MRT sum is greater than the composite operator's MRT, then the exception MRT_SUM_GT_PARENT would be raised. If any individual MRT within a decomposition is greater than the composite operator's MRT then the exception MRT_GT_PARENT would be raised. If any component operator of an operator that has an MRT does not have an MRT then the exception MRT_REQUIRED would be raised.

Component operators may have a PERIOD specified or may inherit the PERIOD and MCP of their composite operator [Ref. 1]. These could be very easily checked by a simple traversal of the OPERATORS tree checking each component operator of those composite operators having an MCP or PERIOD. If the MCP of the component operators is not the same as that of the composite operator, the exception MCP_DIFFERENT would be raised. If a component operator did not have an MCP or PERIOD and the composite operator did have a value in this field, then the appropriate value would be assigned to the component operator.

Once these validity checks are added to the procedure VALIDATE_DATA, the Static Scheduler will be assured that it has done all the validity checking that it can do and the resultant prototype will not fail because the Scheduler ignored some vital information.

F. ATOMIC OPERATOR NAMING CONVENTIONS

Operator naming within the package PSDL_READER needs to be modified in order to eliminate the possibility of name conflicts. Operator names within the

PSDL prototype must be unique to avoid loss of information by the Translator and the FILE_PROCESSOR. It is possible that the same Ada module will be used in several places within a prototype, but they may have different timing and control constraints in each place used. Figure 12 shows the atomic operator D used in two separate places in the PSDL prototype's tree decomposition. In order for the Translator and Static Scheduler to be able to distinguish these modules, their operator names must be unique. Operator names must be identical within the Static Scheduler and the Translator in order for their outputs to be compatible so the following method was adopted by both components of the ESS, but is yet to be implemented in the Static Scheduler. The method agreed upon is to have the attribute grammar processor concatenate a composite operator's name with its atomic operator's names separated by an underscore. Using this technique, no two atomic operators will have the same name, yet the Translator and Scheduler are always using the same name to refer to the same operator. The two instances of Operator D in Figure 12 would then be named B_D and C_D respectively. In order for the Static Scheduler to cooperate as desired with the Translator, the package PSDL_READER must be modified to include this naming convention.

An alternative method of ensuring unique operator names would be to have a Name Manager built in to the PSDL Editors. If successive retrievals and inclusions of an Ada module into the PSDL prototype were given separate names, this problem would not occur for the Static Scheduler and Translator.

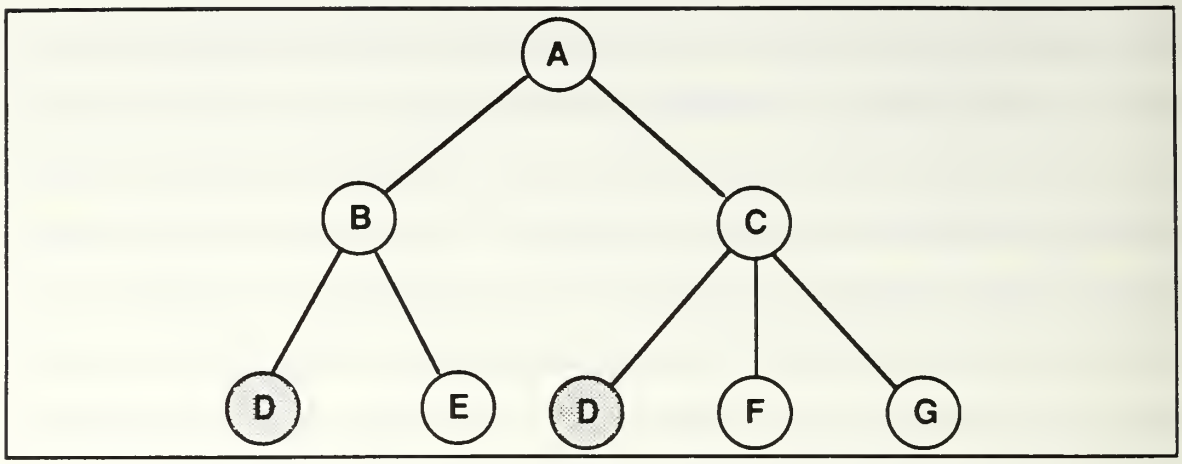


Figure 12. Atomic Operator's Module Used Twice

VI. CONCLUSION

This initial effort at implementation of the Static Scheduler has opened the way for its eventual completion. Generic data structures have been found, implemented and tested that should make completion of the last two packages, HARMONIC_BLOCK_BUILDER and OPERATOR_SCHEDULER, possible. The difficulties identified during this implementation of the first three packages, PSDL_READER, FILE_PROCESSOR, and TOPOLOGICAL_SORTER should be resolved and implemented prior to beginning the completion of the rest of the Static Scheduler because the solutions to the problems identified may have an impact on the final design of the Scheduler or the algorithms within the latter two packages. Implementation of these two packages may uncover other inconsistencies and difficulties with the design, but solutions to most problems can generally be found once the problem has been identified.

Once the Static Scheduler has been completed, it can be integrated into the Execution Support System. Its interfaces with the system are currently confined to three files and an implicit interface with the Translator. The implicit interface is that the operator names uses within the Static Scheduler and the Translator must be the same in order for their outputs to be able to function together. The three files are

- the PSDL source file,
- the NON_CRITS file and
- the Static Schedule file.

The PSDL source file is the Static Scheduler's only input. The NON_CRITS file is a sequential list of non-time critical operator names produces as the Static

Scheduler's first output. The Static Schedule file, when implementation is complete, will contain the Ada source code for the final Static Schedule. These files should remain essentially as they are during integration, so the only modifications necessary should be those required to embed the Static Scheduler within the Debugger. These should be limited to turning the Static Scheduler into a task within the Debugger, and removing the "comment marks" from the few lines of code which allow exception handling to pass the Exception_Operator name to the Debugger.

This thesis has identified several areas for further research during the implementation of the first three packages of the Static Scheduler. These include the following areas identified in Chapter V:

- Implementation of further validity checks including Comparison of OPERATORS and LINKS METs as discussed in section A and implementation of composite to atomic checking on the PERIOD, MCP, and MRT as discussed in section E.
- Removal and replacement of combination composite/atomic link statements by link statements containing only atomic operators as discussed in section B.
- Graph search techniques for finding the maximum path length of MET and MRT values within a network decomposition as discussed in section D.
- Methods to resolve the scheduling conflicts between operators that have precedence relations but no timing or control constraints as discussed in section C.
- Modifying the package PSDL_READER to concatenate atomic operator names to their parent operator's names as discussed in section F.

In addition, further areas for research on the Static Scheduler should include:

- Investigation of how the Static Scheduler should handle data types since these will eventually be implemented in the Translator.
- Implementation of the Static Scheduler for a multiprocessor environment.

The Static Scheduler is crucial to the Computer Aided Prototyping System. A correct and complete implementation is necessary in order for CAPS to function. This thesis has demonstrated that the Static Scheduler can be implemented, but that a correct implementation will not be a trivial accomplishment. Solutions to several minor problems with the design and the one major inconsistency dealing with precedence and non-time critical operators must be found and implemented in order for CAPS to be possible.

CAPS is a realistic alternative method for large embedded software system design, and its realization is within sight, even if this realization consists of only a subset of the Prototyping Description Language as originally envisioned in [Ref.1]. CAPS will be a significant contribution to improvement of the design process, particularly in design feasibility testing, prototype demonstration, and cost control.

LIST OF REFERENCES

1. Luqi, *Rapid Prototyping for Large Software System Design*, Ph.D. Thesis, University of Minnesota, Duluth, Minnesota, May 1986.

2. Raum, H., *The Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

3. Galik, D., *A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

4. O'Hern, J., *A Conceptual Design of a Static Scheduler For Hard Real-Time Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

5. Janson, D., *A Static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

6. Luqi, Berzins, V., Yeh, R., "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, pp. 1409-1423, October 1988.

7. Berzins, V., Luqi, "Semantics of a Real-Time Language," paper presented at *The Proceedings of the Real-Time Systems Symposium*, Huntsville, Alabama, December 1988.

8. Altizer, C., *Implementation of a Language Translator for a Computer Aided Rapid Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

9. Wood, M., *Runtime Support for Rapid Prototyping*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

10. Knuth, D., "Semantics of a Context-Free Language," *Mathematic System Theory*, v. 2, n. 2, pp. 127-145, June 1968.

11. Herndon, R., *The Incomplete AG User's Guide and Reference Manual*, Technical Report 85-37, University of Minnesota, Minneapolis, Minnesota, 1985.

12. Moffitt, C., *Development of a Language Translator for a Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

13. Department of Defense, ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language*, Government Printing Office, Washington, DC, February 14, 1983.

14. Shumate, K., *Understanding Ada*, Harper & Row, Publishers, 1984.

15. Caverly, Phillip and Goldstein, Philip, *Introduction to Ada: A Top-Down Approach for Programmers*, Brooks/Cole Publishing Company, 1986.

16. Miller Jr., E., and Wasserman, A., *High Order Language Evaluation Project Final Report*, February 28, 1977, in Amoroso, S., Wegner, P., Morris, D. and White, D., *Language Evaluation Coordinating Committee Report to the High Order Working Group (HOLWG)*, January 14, 1977.

17. Thorstenson, R., *A Graphical Editor for the Computer Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

18. Britnell, R., *Ada as a Paedeutic Tool for Abstract Data Types*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

APPENDIX A. PROTOTYPE SYSTEM DESCRIPTION LANGUAGE GRAMMAR

The following is the Prototype System Description Language which is used in the Computer Aided Prototyping System.

Optional items are enclosed in square brackets, []. Items that may appear zero or more times appeear in braces, {}. Terminal symbols appear in double quotes, "".

```
Start = psdl

psdl = { component }

component = data_type | operator

data_type = "type" id  type_spec  type_impl

operator = "operator"  id  operator_spec  operator_impl

type_spec = "specification" [type_decl]
        {"operator"  id  operator_spec}
        [functionality]  "end"

type_impl = "implementation"  "ada"  id "{" text "}"
      | "implementation" type_name
        {"operator" id operator_impl}  "end"

operator_spec = "specification"  {interface}
          [functionality] "end"

operator_impl = "implementation" "ada" id "{" text "}"
        |  "implementation" psdl_impl

type_decl = id_list ":" type_name {"," id_list ":" type_name}

functionality = [keywords]  [informal_desc]  [formal_desc]
```

```
psdl_impl = data_flow_diagram  [streams]  [timers]
        [control_constraints] [informal_desc] "end"

type_name =  id "["  type_decl  "]"
      | id

interface = attribute  [reqmts_trace]

id_list = id  {"," id }

keywords = "keywords"  id_list

informal_desc = "description" "{" text "}"

formal_desc = "axioms" "{" text "}"

data_flow_diagram = "graph"  {link}

streams = "data stream" type_decl

timers = "timer" id_list


attribute = generic_param
      | input
      | output
      | states
      | exceptions
      | timing_info

generic_param = "generic" type_decl

input = "input" type_decl

output = "output" type_decl

states = "states" type_decl "initially" expression_list

exceptions = "exceptions"  id_list

timing_info = ["maximum execution time"  time]
        ["minimum calling period" time]
```

```
        ["maximum response time"  time]

reqmts_trace = "by requirements"  id_list

link = id  "."  id  [":" time]  "->"  id

control_constraints = "control constraints"  {constraint}

constraint = "operator"  id
        ["triggered" [trigger] ["if" predicate] [reqmts_trace]]
        ["period" time [reqmts_trace]]
        ["finish within" time [reqmts_trace]]
        {constraint_options}


trigger = "by all"  id_list
    | "by some"  id_list

constraint_options = "output" id_list "if" predicate [reqmts_trace]
            | "exception" id ["if" predicate] [reqmts_trace]
            | timer_op id ["if" predicate] [reqmts_trace]

timer_op = "read timer"
      |  "reset timer"
      |  "start timer"
      |  "stop timer"

expression_list = expression  {"," expression}

time = integer [unit]

unit =  "ms" | "sec" | "min" | "hours"

expression = constant
      | id
      | type_name "." id "(" expression_list ")"

predicate = relation {bool_op relation}

relation = simple_expression
      | simple_expression  rel_op  simple_expression

simple_expression = [sign] integer [unit]
```

```
        | [sign] real
        | ["not"] id
        | string
        | ["not"] "(" predicate ")"
        | ["not"] boolean_constant

bool_op = "and" | "or"

rel_op = "<" | "<=" | ">" | ">=" | "=" | "/=" | ":"

real = integer "." integer

integer = digit {digit}

boolean_constant = "true" | "false"

numeric_constant = real | integer

constant = numeric_constant | boolean_constant

sign = "+" | "-"

char = any printable character except "}"

digit = "0 .. 9"

letter = "a .. z" | "A .. Z" | "_"

alpha_numeric = letter | digit

id = letter {alpha_numeric}

string = """ {char} """

text = {char}
```

APPENDIX B. ATTRIBUTE GRAMMAR SOURCE CODE

The following Attribute Grammar Processor source code is used as the package PSDL_READER by the Static Scheduler. It identifies and retrieves operator names and the critical timing and control information necessary to the Static Scheduler.

```
!definitions of lexical classes

%define Digit   :[0-9]
%define Int     :{Digit}+
%define Letter  :[a-zA-Z_]
%define Alpha   :({Letter}|{Digit})
%define Blank   :[ \t\n]
%define Char    :[^{}]
%define Quote   :["]

! definitions of white space

        :{Blank}+

! definitions of compound symbols and keywords

GTE             :">="
LTE             :"<="
NEQV            :"/="
ARROW           :"->"
TYPE            :type|TYPE
OPERATOR        :operator|OPERATOR
SPECIFICATION   :specification|SPECIFICATION
END             :end|END
GENERIC         :generic|GENERIC
INPUT           :input|INPUT
OUTPUT          :output|OUTPUT
STATES          :states|STATES
INITIALLY       :initially|INITIALLY
EXCEPTIONS      :exceptions|EXCEPTIONS
MAX_EXEC_TIME   :maximum[ ]execution[ ]time
        |MAXIMUM[ ]EXECUTION[ ]TIME
MAX_RESP_TIME   :maximum[ ]response[ ]time
```

```
        |MAXIMUM[ ]RESPONSE[ ]TIME
MIN_CALL_PERIOD :minimum[ ]calling[ ]period
        |MINIMUM[ ]CALLING[ ]PERIOD
MS          :ms|MS
SEC         :sec|SEC
MIN         :min|MIN
HOURS       :hours|HOURS|hrs|HRS|hr|HR
BY          :by[ ]requirements|BY[ ]REQUIREMENTS
KEYWORDS    :keywords|KEYWORDS
DESCRIPTION :description|DESCRIPTION
AXIOMS      :axioms|AXIOMS
IMPLEMENTATION :implementation|IMPLEMENTATION
ADA         :ada|Ada|ADA
GRAPH       :graph|GRAPH
DATA_STREAM :data[ ]stream|DATA[ ]STREAM
TIMER       :timer|TIMER
CONTROL     :control[ ]constraints|CONTROL[ ]CONSTRAINTS
TRIGGERED   :triggered|TRIGGERED
ALL         :by[ ]all|BY[ ]ALL
SOME        :by[ ]some|BY[ ]SOME
PERIOD      :period|PERIOD
FINISH      :finish[ ]within|FINISH[ ]WITHIN
EXCEPTION   :exception|EXCEPTION
READ        :read[ ]timer|READ[ ]TIMER
RESET       :reset[ ]timer|RESET[ ]TIMER
START       :start[ ]timer|START[ ]TIMER
STOP        :stop[ ]timer|STOP[ ]TIMER
IF          :if|IF
NOT         :"~"|"not"|"NOT"
AND         :"&"|"and"|"AND"
OR          :"|"|"or"|"OR"
TRUE        :true|TRUE
FALSE       :false|FALSE
ID          :{Letter}{Alpha}*
STRING_LITERAL :{Quote}{Char}*{Quote}
INTEGER_LITERAL :{Int}
REAL_LITERAL :{Int}"."{Int}
TEXT        :"{"{Char}*"}"
```

```
! operator precedences
! %left means group and evaluate from the left

%left  OR;
%left  AND;
%left  NOT;
%left  '<', '>', '=', GTE, LTE, NEQV;
%left  ':';

%%
! attribute declarations for nonterminal symbols

start { trn: string; };
psdl { trn: string; };
component { trn: string; };
data_type { trn: string; };
operator { trn: string; };
type_spec { trn: string; };
type_decl_1_list { trn: string; };
type_decl { trn: string; };
op_spec_0_list { trn: string; };
operator_spec { trn: string; };
interface { trn: string; };
attribute { trn: string; };
time { trn: string; };
unit { value: int; };
id_list { trn: string; };
reqmts_trace { trn: string; };
functionality { trn: string; };
keywords { trn: string; };
informal_desc { trn: string; };
formal_desc { trn: string; };
type_impl { trn: string; };
op_impl_0_list { trn: string; };
operator_impl { trn: string; children: string;};
psdl_impl { trn: string; children: string; };
data_flow_diagram { trn: string; };
link_0_list { trn: string; };
link { trn: string; };
opt_time { trn: string; };
streams { trn: string; };
type_name { trn: string; };
timers { trn: string; };
```

```
control_constraints { trn: string; children: string; };
constraint_options { trn: string; children: string; };
opt_trig { trn: string; };
trigger { trn: string; };
opt_per { trn: string; };
opt_fin_w { trn: string; };
timer_op { trn: string; };
opt_if_predicate { trn: string; };
predicate { trn: string; };
expression_list { trn: string; };
expression { trn: string; };
relation {trn: string; };
simple_expression { trn: string; };
exception_expr {trn: string; };
rel_op {trn: string; };
sign {trn: string; };

!attrbute declarations for terminal symbols

ID{ %text: string; };
TEXT{ %text: string; };
STRING_LITERAL{ %text: string; };
INTEGER_LITERAL{ %text: string; };
REAL_LITERAL{ %text: string; };


%%
!psdl grammar

start
      : psdl
        { %output(psdl.trn); }
      ;

psdl
      : psdl component
        { psdl[1].trn = [psdl[2].trn,component.trn]; }
      |
        { psdl[1].trn = ""; }
      ;

component
      : data_type
```

```
        { component.trn = ""; }
      | operator
        { component.trn = operator.trn; }
      ;

data_type
      : TYPE ID type_spec type_impl
        { data_type.trn = "";  }
      ;

operator
      : OPERATOR  ID operator_spec operator_impl
        { operator.trn = ["LINEAGE","\n",ID.%text,"\n",
                    operator_impl.children,
                    "END LINEAGE","\n",
                    ID.%text,"\n",operator_spec.trn,
                    operator_impl.trn]; }
      ;

type_spec
      : SPECIFICATION type_decl_1_list op_spec_0_list
         functionality END
        { type_spec.trn =  ""; }
      ;

type_decl_1_list
      : type_decl
        { type_decl_1_list.trn = ""; }
      |
        {type_decl_1_list.trn = ""; }
      ;

type_decl
      : id_list ':' type_name
        { type_decl.trn = ""; }
      | id_list ':' type_name ',' type_decl
        { type_decl.trn = ""; }
      ;


op_spec_0_list
      : op_spec_0_list OPERATOR ID operator_spec
       { op_spec_0_list[1].trn = [op_spec_0_list[2].trn,
```

```
                    ID.%text,"\n",
                    operator_spec.trn]; }
    |
      { op_spec_0_list.trn = ""; }
    ;

operator_spec
    : SPECIFICATION interface functionality END
      { operator_spec.trn = interface.trn; }
    ;

interface
    : interface attribute reqmts_trace
      { interface[1].trn = [interface[2].trn,
                  attribute.trn]; }
    |
      {interface.trn = ""; }
    ;

attribute
    : GENERIC type_decl
      { attribute.trn = ""; }
    | INPUT type_decl
      { attribute.trn = ""; }
    | OUTPUT type_decl
      { attribute.trn = ""; }
    | STATES type_decl INITIALLY expression_list
      { attribute.trn = ["STATE","\n",type_decl.trn,
              "ENDSTATE","\n"]; }
    | EXCEPTIONS id_list
      { attribute.trn = ""; }
    | MAX_EXEC_TIME time
      { attribute.trn = ["MET","\n",time.trn,"\n"]; }
    | MIN_CALL_PERIOD time
      { attribute.trn = ["MCP","\n",time.trn,"\n"]; }
    | MAX_RESP_TIME time
      { attribute.trn = ["MRT","\n",time.trn,"\n"]; }
    ;

id_list
    : ID ',' id_list
      { id_list[1].trn = [ID.%text,"\n",
                  id_list[2].trn] ; }
```

```
    | ID
      { id_list[1].trn = [ID.%text,"\n"]; }
    ;

time
     : INTEGER_LITERAL unit
      { time.trn =
        i2s(s2i(INTEGER_LITERAL.%text)*unit.value); }
    ;

unit
    : MS
      { unit.value = 1000; }
    | SEC
      { unit.value = 1000000; }
    | MIN
      { unit.value = 60000000; }
    | HOURS
      { unit.value = 3600000000; }
    |
      { unit.value = 1000; }
    ;

reqmts_trace
    : BY id_list
      { reqmts_trace.trn = ""; }
    |
      { reqmts_trace.trn = ""; }
    ;

functionality
     : keywords informal_desc formal_desc
      { functionality.trn = ""; }
    ;

keywords
     : KEYWORDS id_list
      { keywords.trn = "\n"; }
    |
      { keywords.trn = ""; }
    ;
```

```
informal_desc
      : DESCRIPTION TEXT
        { informal_desc.trn = "\n"; }
      |
        {informal_desc.trn = ""; }
      ;

formal_desc
      : AXIOMS TEXT
        { formal_desc.trn = "\n"; }
      |
        { formal_desc.trn = ""; }
      ;

type_impl
      : IMPLEMENTATION ADA ID
        { type_impl.trn = ""; }
      | IMPLEMENTATION type_name op_impl_0_list END
        { type_impl.trn = ""; }
      ;

op_impl_0_list
      : op_impl_0_list OPERATOR  ID operator_impl
        { op_impl_0_list[1].trn = [op_impl_0_list[2].trn,
                        ID.%text,"\n",
                        operator_impl.trn]; }
      |
        { op_impl_0_list[1].trn = ""; }
      ;

operator_impl
      : IMPLEMENTATION ADA ID
        { operator_impl.trn = [ID.%text,"\n"];
          operator_impl.children = ["ATOMIC","\n"]; }
      | IMPLEMENTATION psdl_impl
        { operator_impl.trn = psdl_impl.trn;
          operator_impl.children = psdl_impl.children;}
      ;

psdl_impl
      : data_flow_diagram streams timers
        control_constraints informal_desc END
        { psdl_impl.trn = [data_flow_diagram.trn,
```

```
                    control_constraints.trn];
        psdl_impl.children = control_constraints.children;}

data_flow_diagram
      : GRAPH link_0_list
        { data_flow_diagram.trn = link_0_list.trn; }
      ;

link_0_list
      : link_0_list link
        { link_0_list[1].trn = [link_0_list[2].trn,
                    link.trn]; }
      |
        { link_0_list.trn = ""; }
      ;

link
      : ID '.' ID opt_time ARROW ID
        { link.trn = ["LINK","\n",ID[1].%text,"\n",
              ID[2].%text,"\n",opt_time.trn,
              "\n",ID[3].%text,"\n"]; }
      ;

opt_time
      : ':' time
        { opt_time.trn = time.trn; }
      |
        { opt_time.trn = "0"; }
      ;

streams
      : DATA_STREAM type_decl
        { streams.trn = ""; }
      |
        {streams.trn = ""; }
      ;

type_name
      : ID '[' type_decl ']'
        { type_name.trn = "" ; }
      | ID
        { type_name.trn = "" ; }
      ;
```

```
timers
      : TIMER id_list
        { timers.trn = "" ; }
      |
        {timers.trn = ""; }
      ;

control_constraints
      : CONTROL
        { control_constraints.trn = "";
          control_constraints.children ="";}
      | CONTROL OPERATOR ID opt_trig opt_per
          opt_fin_w constraint_options
        {control_constraints.trn = [ID.%text,'\n",
            opt_per.trn,opt_fin_w.trn,
            constraint_options.trn];
          control_constraints.children = [ID.%text,
            "\n",constraint_options.children]; }
      |
        {control_constraints.trn = "";
         control_constraints.children = "";}
      ;

constraint_options
      : OUTPUT id_list IF predicate reqmts_trace
        constraint_options
        { constraint_options[1].trn =
            constraint_options[2].trn;
          constraint_options.children =
            constraint_options[2].children;}
      | EXCEPTION ID opt_if_predicate reqmts_trace
        constraint_options
        { constraint_options[1].trn =
            constraint_options[2].trn;
          constraint_options.children =
            constraint_options[2].children;}
      | timer_op ID opt_if_predicate reqmts_trace
        constraint_options
        { constraint_options[1].trn =
            constraint_options[2].trn;
          constraint_options.children =
            constraint_options[2].children;}
```

```
    | OPERATOR ID opt_trig opt_per opt_fin_w
      constraint_options
      { constraint_options[1].trn =
            [ID.%text,"\n",opt_per.trn,
            opt_fin_w.trn,
            constraint_options[2].trn];
        constraint_options.children = [ID.%text,"\n",
            constraint_options[2].children];}
    |
      { constraint_options.trn = "";
        constraint_options.children = "";}
    ;

opt_trig
    : TRIGGERED trigger opt_if_predicate reqmts_trace
      { opt_trig.trn = ""; }
    |
      {opt_trig.trn = ""; }
    ;

trigger
    : ALL id_list
      { trigger.trn = ""; }
    | SOME id_list
      { trigger.trn = ""; }
    |
      {trigger.trn = ""; }
    ;

opt_per
    :PERIOD time reqmts_trace
      { opt_per.trn = ["PERIOD","\n",time.trn,"\n"]; }
    |
      { opt_per.trn = ""; }
    ;

opt_fin_w
    : FINISH time reqmts_trace
      { opt_fin_w.trn = ["WITHIN","\n",time.trn,"\n"]; }
    |
      { opt_fin_w.trn = ""; }
    ;
```

```
timer_op
      : READ
        { timer_op.trn = ""; }
      | RESET
        { timer_op.trn = ""; }
      | START
        { timer_op.trn = ""; }
      | STOP
        { timer_op.trn = ""; }
      ;

opt_if_predicate
      : IF predicate
        { opt_if_predicate.trn = ""; }
      |
        {opt_if_predicate.trn = ""; }
      ;

expression_list
      : expression
        {expression_list.trn = ""; }
      | expression ',' expression_list
        {expression_list[1].trn = ""; }
      ;

expression
      : INTEGER_LITERAL
        {expression.trn = ""; }
      | REAL_LITERAL
        {expression.trn = ""; }
      | STRING_LITERAL
        {expression.trn = ""; }
      | TRUE
        {expression.trn = ""; }
      | FALSE
        {expression.trn = ""; }
      | ID
        {expression.trn = ""; }
       | type_name '.' ID '(' expression_list ')'
        {expression.trn = ""; }
      ;
```

```
predicate
      : relation
        {predicate.trn = ""; }
      | relation AND predicate
        {predicate[1].trn = ""; }
      | relation OR predicate
        {predicate[1].trn = ""; }
      ;

relation
      : simple_expression rel_op simple_expression
        {relation.trn = ""; }
      | simple_expression
        {relation.trn = ""; }
      ;

simple_expression
      : sign INTEGER_LITERAL unit
        {simple_expression.trn = ""; }
      | sign REAL_LITERAL
        {simple_expression.trn = ""; }
      | ID
        {simple_expression.trn = ""; }
      | STRING_LITERAL
        {simple_expression.trn = ""; }
      | '(' predicate ')'
        {simple_expression.trn = ""; }
      | NOT ID
        {simple_expression.trn = "";}
      | NOT '(' predicate ')'
        {simple_expression.trn = ""; }
      | TRUE
        {simple_expression.trn = ""; }
      | FALSE
        {simple_expression.trn = ""; }
      | NOT TRUE
        {simple_expression.trn = ""; }
      | NOT FALSE
        {simple_expression.trn = ""; }
      ;
```

```
rel_op
      : '<'
        {rel_op.trn = ""; }
      | '>'
        {rel_op.trn = ""; }
      | '='
        {rel_op.trn = ""; }
      | GTE
        {rel_op.trn = ""; }
      | LTE
        {rel_op.trn = ""; }
      | NEQV
        {rel_op.trn = ""; }
      | ':'
        {rel_op.trn = ""; }
      ;

sign
      : '+'
        {sign.trn = ""; }
      | '-'
        {sign.trn = ""; }
      |
        {sign.trn = ""; }
      ;
```

APPENDIX C. IMPLEMENTATIONS OF THE STATIC SCHEDULER'S ABSTRACT DATA TYPES

The following abstract data type implementations are used as the data structures within the Static Scheduler.

```
--  This variable length string abstract data type
-- is used throughout the Static Scheduler.
-- USAGE: with VSTRINGS;
-- package package_name is new VSTRINGS(maximum_length);

with TEXT_IO; use TEXT_IO;
generic
 LAST : NATURAL;
package VSTRINGS is

 subtype STRINDEX is NATURAL;
 FIRST : constant STRINDEX := STRINDEX'FIRST + 1;
 type VSTRING is private;
 NUL : constant VSTRING;

-- Attributes of a VSTRING

 function LEN(FROM : VSTRING) return STRINDEX;
 function MAX(FROM : VSTRING) return STRINDEX;
 function STR(FROM : VSTRING) return STRING;
 function CHAR(FROM: VSTRING;
          POSITION : STRINDEX := FIRST)
          return CHARACTER;

-- Comparisons

 function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
 function ">" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
 function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
 function ">=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
 function equal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
 function notequal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
```

```
-- Input/Output

  procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING);
  procedure PUT(ITEM : in VSTRING);
  procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING);
  procedure PUT_LINE(ITEM : in VSTRING);
  procedure GET(FILE : in FILE_TYPE; ITEM : out VSTRING;
                LENGTH : in STRINDEX := LAST);
  procedure GET(ITEM : out VSTRING;
                LENGTH : in STRINDEX := LAST);
  procedure GET_LINE(FILE : in FILE_TYPE; ITEM : in out VSTRING);
  procedure GET_LINE(ITEM : in out VSTRING);

-- Extraction

  function SLICE(FROM: VSTRING;FRONT, BACK : STRINDEX)
                 return VSTRING;
  function SUBSTR(FROM: VSTRING;START, LENGTH: STRINDEX)
                  return VSTRING;
  function DELETE(FROM: VSTRING;FRONT, BACK : STRINDEX)
                  return VSTRING;

-- Editing

  function INSERT(TARGET: VSTRING; ITEM: VSTRING;
                  POSITION: STRINDEX := FIRST) return VSTRING;
  function INSERT(TARGET: VSTRING; ITEM: STRING;
                  POSITION: STRINDEX := FIRST) return VSTRING;
  function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
                  POSITION: STRINDEX := FIRST) return VSTRING;

  function APPEND(TARGET: VSTRING; ITEM: VSTRING;
                  POSITION: STRINDEX) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: STRING;
                  POSITION: STRINDEX) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: CHARACTER;
                  POSITION: STRINDEX) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: VSTRING) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: STRING) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: CHARACTER)
                  return VSTRING;

  function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
```

```
                        POSITION: STRINDEX := FIRST) return VSTRING;
  function REPLACE(TARGET: VSTRING; ITEM: STRING;
                        POSITION: STRINDEX := FIRST) return VSTRING;
  function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
                        POSITION: STRINDEX := FIRST) return VSTRING;

-- Concatenation

  function "&" (LEFT: VSTRING; RIGHT : VSTRING) return VSTRING;
  function "&" (LEFT: VSTRING; RIGHT : STRING) return VSTRING;
  function "&" (LEFT: VSTRING; RIGHT : CHARACTER) return VSTRING;
  function "&" (LEFT: STRING; RIGHT : VSTRING) return VSTRING;
  function "&" (LEFT: CHARACTER; RIGHT : VSTRING) return VSTRING;

-- Determine the position of a substring

  function INDEX(WHOLE: VSTRING; PART: VSTRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;
  function INDEX(WHOLE : VSTRING; PART : STRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;
  function INDEX(WHOLE : VSTRING; PART : CHARACTER;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;
  function RINDEX(WHOLE: VSTRING; PART: VSTRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;
  function RINDEX(WHOLE : VSTRING; PART : STRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;
  function RINDEX(WHOLE : VSTRING; PART : CHARACTER;
                        OCCURRENCE : NATURAL := 1) return STRINDEX;

-- Conversion from other associated types

  function VSTR(FROM : STRING) return VSTRING;
  function VSTR(FROM : CHARACTER) return VSTRING;
  function "+" (FROM : STRING) return VSTRING;
  function "+" (FROM : CHARACTER) return VSTRING;

  generic
    type FROM is private;
    type TO is private;
    with function STR(X : FROM) return STRING is <>;
    with function VSTR(Y : STRING) return TO is <>;
    function CONVERT(X : FROM) return TO;
```

```
  private
   type VSTRING is
    record
     LEN : STRINDEX := STRINDEX'FIRST;
     VALUE : STRING(FIRST..LAST) := (others=>ASCII.NUL);
    end record;

   NUL : constant VSTRING := (STRINDEX'FIRST,
                    (others => ASCII.NUL));
end VSTRINGS;


package body VSTRINGS is
 -- local declarations
 FILL_CHAR : constant CHARACTER := ASCII.NUL;

 procedure FORMAT(THE_STRING : in out VSTRING;
                            OLDLEN : in STRINDEX := LAST) is
  -- fill string with FILL_CHAR to null out old values
  begin -- FORMAT (Local Procedure)
   THE_STRING.VALUE(THE_STRING.LEN + 1..OLDLEN) :=
                                  others => FILL_CHAR);
  end FORMAT;

 -- bodies of visible operations

 function LEN(FROM : VSTRING) return STRINDEX is
  begin -- LEN
   return(FROM.LEN);
  end LEN;

 function MAX(FROM : VSTRING) return STRINDEX is
  begin -- MAX
   return(LAST);
  end MAX;

 function STR(FROM : VSTRING) return STRING is
  begin -- STR
   return(FROM.VALUE(FIRST .. FROM.LEN));
  end STR;

 function CHAR(FROM : VSTRING; POSITION : STRINDEX := FIRST)
                    return CHARACTER is
```

```
 begin -- CHAR
  if POSITION not in FIRST .. FROM.LEN
   then raise CONSTRAINT_ERROR;
  end if;
  return(FROM.VALUE(POSITION));
 end CHAR;

function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- "<"
  return(LEFT.VALUE < RIGHT.VALUE);
 end "<";

function ">" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- ">"
  return(LEFT.VALUE > RIGHT.VALUE);
 end ">";

function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- "<="
  return(LEFT.VALUE <= RIGHT.VALUE);
 end "<=";

function ">=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- ">="
  return(LEFT.VALUE >= RIGHT.VALUE);
 end ">=";

function equal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- equal
  return(LEFT.VALUE = RIGHT.VALUE);
 end equal;

function notequal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
 begin -- notequal
  return(LEFT.VALUE /= RIGHT.VALUE);
 end notequal;

procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING) is
 begin -- PUT
  PUT(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
 end PUT;

procedure PUT(ITEM : in VSTRING) is
```

```
 begin -- PUT
   PUT(ITEM.VALUE(FIRST .. ITEM.LEN));
 end PUT;

procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING) is
 begin -- PUT_LINE
   PUT_LINE(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
 end PUT_LINE;

procedure PUT_LINE(ITEM : in VSTRING) is
 begin -- PUT_LINE
   PUT_LINE(ITEM.VALUE(FIRST .. ITEM.LEN));
 end PUT_LINE;

procedure GET(FILE : in FILE_TYPE; ITEM : out VSTRING;
                    LENGTH : in STRINDEX := LAST) is
 begin -- GET
  if LENGTH not in FIRST .. LAST
    then raise CONSTRAINT_ERROR;
   end if;
  ITEM := NUL;
  for INDEX in FIRST .. LENGTH loop
    GET(FILE, ITEM.VALUE(INDEX));
    ITEM.LEN := INDEX;
   end loop;
 end GET;

procedure GET(ITEM : out VSTRING; LENGTH : in STRINDEX := LAST) is
 begin -- GET
  if LENGTH not in FIRST .. LAST
    then raise CONSTRAINT_ERROR;
   end if;
  ITEM := NUL;
  for INDEX in FIRST .. LENGTH loop
    GET(ITEM.VALUE(INDEX));
    ITEM.LEN := INDEX;
   end loop;
 end GET;

procedure GET_LINE(FILE : in FILE_TYPE; ITEM : in out VSTRING) is
 OLDLEN : constant STRINDEX := ITEM.LEN;
 begin -- GET_LINE
   GET_LINE(FILE, ITEM.VALUE, ITEM.LEN);
```

```
   FORMAT(ITEM, OLDLEN);
  end GET_LINE;

procedure GET_LINE(ITEM : in out VSTRING) is
  OLDLEN : constant STRINDEX := ITEM.LEN;
  begin -- GET_LINE
    GET_LINE(ITEM.VALUE, ITEM.LEN);
   FORMAT(ITEM, OLDLEN);
  end GET_LINE;

function SLICE(FROM : VSTRING; FRONT,BACK : STRINDEX)
          return VSTRING is
  begin -- SLICE
    if ((FRONT not in FIRST..FROM.LEN) or else
      (BACK not in FIRST..FROM.LEN)) and then FRONT <= BACK
     then raise CONSTRAINT_ERROR;
    end if;
    return(Vstr(FROM.VALUE(FRONT .. BACK)));
  end SLICE;

function SUBSTR(FROM : VSTRING; START, LENGTH : STRINDEX)
                      return VSTRING is
  begin -- SUBSTR
    if (START not in FIRST .. FROM.LEN) or else
      ((START + LENGTH - 1 not in FIRST .. FROM.LEN)
       and then (LENGTH > 0))
     then raise CONSTRAINT_ERROR;
    end if;
    return(Vstr(FROM.VALUE(START .. START + LENGTH -1)));
  end SUBSTR;

function DELETE(FROM : VSTRING; FRONT, BACK : STRINDEX)
                      return VSTRING is
  TEMP : VSTRING := FROM;
  begin -- DELETE
    if ((FRONT not in FIRST..FROM.LEN) or else
      (BACK not in FIRST..FROM.LEN)) and then FRONT <= BACK
     then raise CONSTRAINT_ERROR;
    end if;
    if FRONT > BACK then return(FROM); end if;
    TEMP.LEN := FROM.LEN - (BACK - FRONT) - 1;
    TEMP.VALUE(FRONT .. TEMP.LEN) :=
          FROM.VALUE(BACK + 1 .. FROM.LEN);
```

```
   FORMAT(TEMP, FROM.LEN);
   return(TEMP);
  end DELETE;

function INSERT(TARGET: VSTRING; ITEM: VSTRING;
                        POSITION : STRINDEX := FIRST)
                        return VSTRING is
  TEMP : VSTRING;
  begin -- INSERT
   if POSITION not in FIRST .. TARGET.LEN
    then raise CONSTRAINT_ERROR;
   end if;
   if TARGET.LEN + ITEM.LEN > LAST
    then raise CONSTRAINT_ERROR;
    else TEMP.LEN := TARGET.LEN + ITEM.LEN;
   end if;
   TEMP.VALUE(FIRST .. POSITION - 1) :=
          TARGET.VALUE(FIRST .. POSITION - 1);
   TEMP.VALUE(POSITION .. (POSITION + ITEM.LEN - 1)) :=
    ITEM.VALUE(FIRST .. ITEM.LEN);
   TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
    TARGET.VALUE(POSITION .. TARGET.LEN);
   return(TEMP);
  end INSERT;

function INSERT(TARGET: VSTRING; ITEM: STRING;
                        POSITION : STRINDEX := FIRST)
                        return VSTRING is
  begin -- INSERT
   return INSERT(TARGET, VSTR(ITEM), POSITION);
  end INSERT;

function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
                        POSITION : STRINDEX := FIRST)
                        return VSTRING is
  begin -- INSERT
   return INSERT(TARGET, VSTR(ITEM), POSITION);
  end INSERT;

function APPEND(TARGET: VSTRING; ITEM: VSTRING;
                         POSITION : STRINDEX) return VSTRING is
  TEMP : VSTRING;
  POS : STRINDEX := POSITION;
```

```
 begin -- APPEND
  if POSITION not in FIRST .. TARGET.LEN
   then raise CONSTRAINT_ERROR;
  end if;
  if TARGET.LEN + ITEM.LEN > LAST
   then raise CONSTRAINT_ERROR;
   else TEMP.LEN := TARGET.LEN + ITEM.LEN;
  end if;
  TEMP.VALUE(FIRST .. POS) := TARGET.VALUE(FIRST .. POS);
  TEMP.VALUE(POS + 1 .. (POS + ITEM.LEN)) :=
        ITEM.VALUE(FIRST .. ITEM.LEN);
  TEMP.VALUE((POS + ITEM.LEN + 1) .. TEMP.LEN) :=
   TARGET.VALUE(POS + 1 .. TARGET.LEN);
  return(TEMP);
 end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING;
                POSITION : STRINDEX) return VSTRING is
 begin -- APPEND
  return APPEND(TARGET, VSTR(ITEM), POSITION);
 end APPEND;

function APPEND(TARGET: VSTRING; ITEM: CHARACTER;
                POSITION : STRINDEX) return VSTRING is
 begin -- APPEND
  return APPEND(TARGET, VSTR(ITEM), POSITION);
 end APPEND;

function APPEND(TARGET: VSTRING; ITEM: VSTRING)
                return VSTRING is
 begin -- APPEND
  return(APPEND(TARGET, ITEM, TARGET.LEN));
 end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING)
                return VSTRING is
 begin -- APPEND
  return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
 end APPEND;

function APPEND(TARGET: VSTRING; ITEM: CHARACTER)
                return VSTRING is
 begin -- APPEND
```

```
    return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
  end APPEND;

function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
                          POSITION : STRINDEX := FIRST)
                          return VSTRING is
  TEMP : VSTRING;
  begin -- REPLACE
    if POSITION not in FIRST .. TARGET.LEN
     then raise CONSTRAINT_ERROR;
    end if;
    if POSITION + ITEM.LEN - 1 <= TARGET.LEN
     then TEMP.LEN := TARGET.LEN;
     elsif POSITION + ITEM.LEN - 1 > LAST
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := POSITION + ITEM.LEN - 1;
    end if;
    TEMP.VALUE(FIRST .. POSITION - 1) :=
          TARGET.VALUE(FIRST .. POSITION - 1);
    TEMP.VALUE(POSITION..(POSITION + ITEM.LEN - 1)) :=
     ITEM.VALUE(FIRST .. ITEM.LEN);
    TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
     TARGET.VALUE((POSITION + ITEM.LEN) .. TARGET.LEN);
    return(TEMP);
  end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: STRING;
                          POSITION : STRINDEX := FIRST)
                          return VSTRING is
  begin -- REPLACE
    return REPLACE(TARGET, VSTR(ITEM), POSITION);
  end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
                          POSITION : STRINDEX := FIRST)
                          return VSTRING is
  begin -- REPLACE
    return REPLACE(TARGET, VSTR(ITEM), POSITION);
  end REPLACE;

function "&"(LEFT:VSTRING; RIGHT : VSTRING) return VSTRING is
  TEMP : VSTRING;
  begin -- "&"
```

```
   if LEFT.LEN + RIGHT.LEN > LAST
    then raise CONSTRAINT_ERROR;
    else TEMP.LEN := LEFT.LEN + RIGHT.LEN;
   end if;
   TEMP.VALUE(FIRST .. TEMP.LEN) :=
          LEFT.VALUE(FIRST .. LEFT.LEN) &
          RIGHT.VALUE(FIRST .. RIGHT.LEN);
  return(TEMP);
 end "&";

function "&"(LEFT:VSTRING; RIGHT : STRING) return VSTRING is
 begin -- "&"
  return LEFT & VSTR(RIGHT);
 end "&";

function "&"(LEFT:VSTRING; RIGHT : CHARACTER) return VSTRING is
 begin -- "&"
  return LEFT & VSTR(RIGHT);
 end "&";

function "&"(LEFT : STRING; RIGHT : VSTRING) return VSTRING is
 begin -- "&"
  return VSTR(LEFT) & RIGHT;
 end "&";

function "&"(LEFT : CHARACTER; RIGHT : VSTRING) return VSTRING is
 begin -- "&"
  return VSTR(LEFT) & RIGHT;
 end "&";

Function INDEX(WHOLE : VSTRING; PART : VSTRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX is
 NOT_FOUND : constant NATURAL := 0;
 INDEX : NATURAL := FIRST;
 COUNT : NATURAL := 0;
 begin -- INDEX
  if PART = NUL then return(NOT_FOUND); -- by definition
   end if;
  while INDEX + PART.LEN - 1 <= WHOLE.LEN and then
     COUNT < OCCURRENCE loop
    if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
      PART.VALUE(1 .. PART.LEN)
     then COUNT := COUNT + 1;
```

```
    end if;
    INDEX := INDEX + 1;
   end loop;
   if COUNT = OCCURRENCE
    then return(INDEX - 1);
    else return(NOT_FOUND);
   end if;
  end INDEX;

Function INDEX(WHOLE : VSTRING; PART : STRING;
                        OCCURRENCE : NATURAL := 1) return STRINDEX is
  begin -- Index
   return(Index(WHOLE, VSTR(PART), OCCURRENCE));
  end INDEX;

Function INDEX(WHOLE : VSTRING; PART : CHARACTER;
                        OCCURRENCE : NATURAL := 1) return STRINDEX is
  begin -- Index
   return(Index(WHOLE, VSTR(PART), OCCURRENCE));
  end INDEX;

function RINDEX(WHOLE: VSTRING; PART:VSTRING;
                         OCCURRENCE:NATURAL := 1)  return STRINDEX is
  INDEX : INTEGER := WHOLE.LEN - (PART.LEN -1);
  COUNT : NATURAL := 0;
  begin -- RINDEX
   if PART = NUL then
     return(NOT_FOUND); -- by definition
   end if;
   while INDEX >= FIRST and then COUNT < OCCURRENCE loop
    if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
      PART.VALUE(1 .. PART.LEN)
     then COUNT := COUNT + 1;
    end if;
    INDEX := INDEX - 1;
   end loop;
   if COUNT = OCCURRENCE
    then
     if COUNT > 0
       then return(INDEX + 1);
       else return(NOT_FOUND);
      end if;
    else return(NOT_FOUND);
```

```
    end if;
  end RINDEX;

Function RINDEX(WHOLE : VSTRING; PART : STRING;
                          OCCURRENCE : NATURAL := 1) return STRINDEX is
  begin -- Rindex
    return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
  end RINDEX;

Function RINDEX(WHOLE : VSTRING; PART : CHARACTER;
                          OCCURRENCE : NATURAL := 1) return STRINDEX is
  begin -- Rindex
    return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
  end RINDEX;

function VSTR(FROM : CHARACTER) return VSTRING is
  TEMP : VSTRING;
  begin -- VSTR
    if LAST < 1
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := 1;
    end if;
    TEMP.VALUE(FIRST) := FROM;
    return(TEMP);
  end VSTR;

function VSTR(FROM : STRING) return VSTRING is
  TEMP : VSTRING;
  begin -- VSTR
    if FROM'LENGTH > LAST
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := FROM'LENGTH;
    end if;
    TEMP.VALUE(FIRST .. FROM'LENGTH) := FROM;
    return(TEMP);
  end VSTR;

Function "+" (FROM : STRING) return VSTRING is
  begin -- "+"
    return(VSTR(FROM));
  end "+";

Function "+" (FROM : CHARACTER) return VSTRING is
```

```
    begin
     return(VSTR(FROM));
    end "+";

  function CONVERT(X : FROM) return TO is
    begin -- CONVERT
      return(VSTR(STR(X)));
    end CONVERT;
end VSTRINGS;

-- DISTRIBUTION AND COPYRIGHT:
--
-- This software is released to the Public Domain
-- (note: software released to the Public Domain
-- is not subject to copyright protection).
-- Restrictions on use or distribution:  NONE
```

```
-- This package defines the operations available
-- on the linked list abstract data type which is
-- used in the Static Scheduler.

Generic
  Type Item is Private;

Package Generic_List is
  Type List is Private;
  BEYOND_END : Exception;
  NOT_FOUND : Exception;
  INSERT_BEYOND : Exception;
  DELETE_OUT_OF_RANGE : Exception;

  Procedure Clear (L : in out List);
  -- pre - None.
  -- post - L-pre exists as an empty list.

  Function Full(L : in List) Return Boolean;
  -- pre - None.
  -- post - True if list L cannot have more items added,
  --        otherwise False.

  Function Empty(L : in List) Return Boolean;
  -- pre - None.
  -- post - True if list L has no items in it,
  -- otherwise False.

  Function Member(L: in List; I: in Item) Return Boolean;
  -- pre - None.
  -- post - True if list L contains I, otherwise False.

  Procedure Insert (L : in out List; P : in Integer; I : in Item);
  -- pre - The size of L has not reached its maximum.
  -- post - L includes item I in the Pth place
  -- exception raised
  -- - INSERT_BEYOND if P > (Length of list + 1)

  Procedure Delete (L : in out List; P : in Integer; I : out Item);
  -- pre - L is not empty.
  -- post - I was the Pth item of the List.
  --   L no longer contains I.
  -- exception raised
```

```
   -- - DELETE_OUT_OF_RANGE if P > Length of L.

   Procedure Length (L : in out List; Long : out Integer);
   -- pre - L exists.
   -- post - Long is equal to the number of items in L.

   Procedure Find_Item (L : in out List; P : in Integer; I : out Item);
   -- pre - L is not empty.
   -- post - I is the Pth item of L.  L is unchanged.
   -- exception raised - BEYOND_END if P > Length of List.

   Procedure Find_Pos (L : in out List; P : out Integer; I : in Item);
   -- pre - L is not empty.
   -- post - P is the position of I in  L.
   -- exception raised
   -- - NOT_FOUND if I is not found in the List.

   Private
     Type Node;
     Type List is Access Node;

end Generic_List;


with Unchecked_Deallocation;
package body Generic_List is

   Type Node is
     Record
       Element : Item;
       Next : List;
     end Record;

   Procedure Return_Node is new Unchecked_Deallocation(Node,List);

   Procedure Clear (L : in out List) is
   -- post - L-pre exists as an empty list.
   Temp_Ptr : List;
   begin
     If not Empty(L) then
       While (L.Next /= null) -- Reclaims each node
       Loop                   -- except last one.
         Temp_Ptr := L;
```

```
      L := L.Next;
      Return_Node (Temp_Ptr);
    end Loop;
    Return_Node (L);       -- Reclaims last node in list.
  end If;
end Clear;

Function FULL(L : in List) Return Boolean is
-- post - True if the list L can not have more items added,
--        otherwise False.
Temp_Ptr : List;
begin
  Temp_Ptr := new Node;      -- Generates new pointer
  Return_Node(Temp_Ptr);     -- Returns pointer to memory
  Return (false);
Exception
  when STORAGE_ERROR =>
    Return (true);          -- Out of memory.
  when others =>
    raise;
end FULL;

Function Empty(L : in List) Return Boolean is
-- post - True if list L has no items in it,
--        otherwise False.
begin
  Return (L = null);
end Empty;

Function Member(L : in List; I: in Item) Return Boolean is
-- post - True if list L contains I, otherwise False.
Temp_Ptr: List;
begin
  if not Empty(L) then
    Temp_Ptr := L;
    while (Temp_Ptr.Next /= null)
    loop
      if (Temp_Ptr.Element = I) then
        return TRUE;
      end if;
      Temp_Ptr := Temp_Ptr.Next;
    end loop;
    return False;
```

```
  else
    return False;
  end if;
end Member;

Procedure Insert (L : in out List; P : in Integer; I : in Item) is
-- pre - The size of L has not reached its maximum.
-- post - L includes item I in the Pth place
-- exceptions raised
-- - INSERT_BEYOND if P > (Length of list + 1).
Before_New : List;
Temp_Ptr : List;
New_Ptr : List;
Previous : Integer;
NumItems : Integer;
begin
  New_Ptr := new Node'(I,null); -- node to be inserted.
  Previous := P;
  Temp_Ptr := L;
  If P = 1 then          -- Inserting at front of list.
    New_Ptr.Next := L;
    L := New_Ptr;
  else
    If Temp_Ptr = null then
      raise INSERT_BEYOND;
    end If;
    While (Previous /= 1) -- Before_New will be pointing to
    Loop                  -- item in list that will precede
      Previous := Previous - 1; -- new item being inserted.
      Before_New := Temp_Ptr;
      Temp_Ptr := Temp_Ptr.Next; -- Temp_Ptr points to item
                    -- that succeeds new item in list.
      If Temp_Ptr = null then
        Exit;
      end If;
    end Loop;
    If Previous = 1 then
      Before_New.Next := New_Ptr;
      New_Ptr.Next := Temp_Ptr;
    else
      raise INSERT_BEYOND;
    end If;
  end if;
```

```
end Insert;

Procedure Delete (L : in out List; P : in Integer; I : out Item) is
-- pre - L is not empty.
-- post - I was the Pth item of the List.
--        L no longer contains I.
-- exceptions raised
-- - DELETE_OUT_OF_RANGE if P > the length of L.
Temp_Ptr : List;
Node_Before : List;
NumItems : Integer;
begin
  Temp_Ptr := L;
  If P = 1 then  -- Deletion if first item in list.
    I := Temp_Ptr.Element;
    L := Temp_Ptr.Next;
    Return_Node(Temp_Ptr);
  else
    For Count in 1..(P-1)
    -- Node_Before will point to item in
    -- list before the one to be deleted.
    Loop
      Node_Before := Temp_Ptr;
      If Temp_Ptr = null then
        raise DELETE_OUT_OF_RANGE;
         -- Can't delete beyond end of list.
      end If;
      Temp_Ptr := Temp_Ptr.Next;
      -- Temp_Ptr will point to item to be deleted.
    end Loop;
    If Temp_Ptr = null then
      raise DELETE_OUT_OF_RANGE;
       -- Can't delete beyond end of list.
    end If;
    Node_Before.Next := Temp_Ptr.Next;
    I := Temp_Ptr.Element; -- Content of node.
    Return_Node(Temp_Ptr); -- Reclaims pointer.
  end if;
end Delete;

Procedure Length (L : in out List; Long : out Integer) is
  -- pre - L exists.
  -- post - Long is equal to the number of items in L.
```

```
Temp_Ptr : List;
Count : Integer;
begin
  if Empty(L) then
    Long := 0;              -- Returns length.
  else
    Temp_Ptr := L;
    Count := 1;
    While (Temp_Ptr.Next /= null)
      -- Traverse list incrementing count.
    Loop
      Count := Count + 1;
      Temp_Ptr := Temp_Ptr.Next;
    end Loop;
    Long := Count;          -- Returns length.
  end if;
end Length;

Procedure Find_Item (L : in out List; P : in Integer; I : out Item) is
-- pre - L is not empty.
-- post - I is the Pth item of L.  L is unchanged.
-- exception raised
--  - BEYOND_END if P > length of the list.
NumItems : Integer;
Temp_Ptr : List;
begin
  Temp_Ptr := L;
  For Count in 1..(P-1)  -- Traverse list to the Pth item.
  Loop
    Temp_Ptr := Temp_Ptr.Next;
    If Temp_Ptr = null then
      raise BEYOND_END; -- Can't find Pth item when list
    end If;             -- length is less than P.
  end Loop;
  I := Temp_Ptr.Element;  -- Returns the Pth item.
end Find_Item;

Procedure Find_Pos (L : in out List; P : out Integer; I : in Item) is
-- pre - L is not empty.
-- post - P is the position of I in  L.
-- exception raised
--  - NOT_FOUND if I is not found in the list.
Temp_Ptr : List;
```

```
    New_Ptr : List;
    Previous : List;
    Count : Integer;
    begin
      Temp_Ptr := L;
      If Temp_Ptr.Element = I then -- First item in the list.
        P := 1;
      else
        Count := 1;
        While (Temp_Ptr.Element /= I) and
            (Temp_Ptr.Next /= null)
         -- Traverse list until found or end of the list.
        Loop
           Temp_Ptr := Temp_Ptr.Next;
          Count := Count + 1;    -- Count each node checked.
        end Loop;
        If Temp_Ptr.Element /= I then
          raise NOT_FOUND; -- Item desired is not in the list.
        else
          P := Count;      -- Item located in the Pth position.
        end if;
      end if;
    end Find_Pos;

end Generic_List;
```

```
-- This generic package is used by the N_ary tree package
-- that is used by the Static Scheduler.

generic
 type LISTTOKEN is private;

package LISTS is
 type LIST is private;
 LIST_END: exception;

  procedure FindNext (L : in  LIST);
  --   Makes current node's successor the current node.
  --   If current node is last in list raises END_LIST.

  procedure FindPrior (L : in  LIST);
  --   precon  - The current element in the list is not
  --   the first element and the list is not empty.
  --   Makes current node's predecessor the current node.

  procedure FindIth (L : in  LIST; I : in natural);
  -- precon  - 1 <= I <= Size(L)
  -- action  - Makes the i'th node in list the current node.

  procedure FindPosition (L : in  LIST; El : in LISTTOKEN;
                                  POS : out NATURAL);
  -- Searches for 'El' in the list.  If it is found,
  -- 'El' will be the current list element and POS will
  -- contain the position of 'El' in the list.
  -- If 'El' does not exist in the list POS will return
  -- with the value 0.
  -- If the list is empty, POS will return with the value 0.

  function IsInList (L : in LIST; Item : in LISTTOKEN) return boolean;
  -- Searches for Item in list, returns true if in the list,
  -- false , otherwise.

  procedure Retrieve (L : in LIST; Element : out LISTTOKEN);
  -- precon  - LIST must not be empty.
  -- action  - Retrieves the current node token value  and returns it in Element.

  procedure Update (L : in out LIST; Element: in LISTTOKEN);
  -- precon  - LIST must not be empty.
  -- action  - Places the value of Element into
```

```
   -- the current node.

   procedure InsertBefore (L : in out LIST; Element : in LISTTOKEN);
   -- postcon - Element is now an element of the list and is the current node;
   -- action  - Inserts Element as the predecessor to the current element.

   procedure InsertAfter (L : in out LIST; Element : in LISTTOKEN);
   -- postcon - Element is an element of the list and is  the current node.
   -- action  - Inserts Element as the successor to the current element.

   procedure Delete (L : in out LIST);
   -- precon  - LIST is not empty.
   -- postcon - The previously current node is no longer
   --           in the list. If the list contains a single
   --           element, then that element is current.
   --           If the list contains at least two elements,
   --           then the successor to the previously current
   --           node, if it exists, is current, otherwise the
   --           first element is current.
   --           If the previously current node's predecessor
   --           and successor both exist, then the successor
   --           node is now the successor to the previous node.
   -- action  - Deletes the current list element.

   function Size (L : in LIST) return natural;
   -- postcon - LISTSize is the number of elements in the list.
   -- action  - Returns current number of elements in the list.

   procedure Create (L : in out LIST);
   -- postcon - A new list exists and is empty.
   -- actRon  - Creates a new list.

   procedure Kill (L : in out LIST);
   -- postcon - The list no longer exists.
   -- action  - Terminates the list and returns all allocated storage to the system.

private
  type LISTINSTANCE;
  type LISTNODE;
  type LIST is access LISTINSTANCE;
  type LISTNODEPTR is access LISTNODE;
  type LISTNODE is record
       Tokn  : LISTTOKEN;
```

```
        Left  : LISTNODEPTR;
        Right : LISTNODEPTR;
      end record;
  type LISTINSTANCE is record
        Head          : LISTNODEPTR;
        Current       : LISTNODEPTR;
        CurrentSize   : natural := 0;
        CurrentPosition : natural := 0;
      end record;

end LISTS;

package body LISTS is

  function Size (L : in LIST) return natural is
  -- Get the current number of elements in the list.
  begin
    if L = null then
      return 0;
    else
      return L.CurrentSize;
    end if;
  end Size;

  procedure FindNext (L : in  LIST) is
  -- Make current's successor current
  begin
     if L.Current.Right /= null then
        L.Current := L.Current.Right;
        L.CurrentPosition := L.CurrentPosition + 1;
     else
        raise LIST_END;
     end if;
  end FindNext;

  procedure FindPrior (L : in  LIST) is
  -- Make current's predecessor current
  begin
     L.Current := L.Current.Left;
     L.CurrentPosition := L.CurrentPosition - 1;
  end FindPrior;

  procedure FindIth (L : in  LIST; I : in natural) is
```

```
-- Make ith list element current
begin
   if I > Size(L) then
     raise LIST_END;
   else
     L.Current := L.Head;
     for j in 1 .. I-1 loop
        L.Current := L.Current.Right;
     end loop;
     L.CurrentPosition := I;
   end if;
end FindIth;

procedure FindPosition (L : in  LIST;  El  : in LISTTOKEN;
                                POS : out NATURAL) is
-- Locate El in LIST and make it current.
-- POS records El's position in the LIST.
TempEl  : LISTTOKEN; -- temporary storage of list elements.
begin
 if Size (L) > 0 then
    FindIth (L,1);     -- start at head of the list.
    for j in 1 .. Size(L) loop
       Retrieve (L,TempEl);
        -- look at current element in the LIST.
       exit when TempEl = El;
       FindNext (L);  -- goto next element in the LIST.
    end loop;
    POS := L.CurrentPosition;  -- Current element is the
 else  -- LIST is empty.         desired element.
    POS := 0;
 end if;
exception
  when LIST_END =>   -- El was never located in the LIST.
    POS := 0;
end FindPosition;

function IsInList (L : in LIST; Item : in LISTTOKEN) return boolean is
  TempToken : LISTTOKEN;
  NodeCount : natural := 1;
  TempCurrent : LISTNODEPTR := L.Current;
begin
  If Size(L) = 0 then return false;
  end if;
```

```
 loop
    exit when NodeCount >= L.CurrentSize;
     FindIth(L,NodeCount);
     Retrieve (L,TempToken);
     if TempToken = Item
     then L.Current := TempCurrent;
          return true;
     end if;
     NodeCount := NodeCount + 1;
  end loop;
  L.Current := TempCurrent;
  return false;
end IsInList;

procedure Retrieve (L : in LIST; Element : out LISTTOKEN) is
-- Retrieve current element value.
begin
  Element := L.Current.Tokn;
end Retrieve;

procedure Update (L : in out LIST; Element : in LISTTOKEN) is
-- Change current element value
begin
  L.Current.Tokn := Element;
end Update;

procedure InsertAfter (L : in out LIST;  Element : in LISTTOKEN) is
-- Insert Element as the successor to the current element.
  P : LISTNODEPTR;
begin
    P := new LISTNODE;
    P.Tokn := Element;
    if L.CurrentSize = 0
    then
      L.Head := P;
      L.CurrentPosition := 1;
      P.Right := P;
      P.Left := P;
     else     -- Link node P into the doubly linked list
       L.Current.Right.Left := P;
       P.Right := L.Current.Right;
       P.Left := L.Current;
       L.Current.Right := P;
```

```
      L.CurrentPosition := L.CurrentPosition + 1;
    end if;
    L.CurrentSize := L.CurrentSize + 1;
    L.Current := P;
end InsertAfter;

procedure InsertBefore (L : in out LIST; Element : in LISTTOKEN) is
-- Insert Element as the predecessor to the current
begin
    if L.CurrentSize /= 0
    then
        L.Current := L.Current.Left;
        L.CurrentPosition := L.CurrentPosition - 1;
    end if;
    InsertAfter (L,Element);
    if L.Current.Right = L.Head
    then L.Head := L.Current;
    end if;
end InsertBefore;

procedure Delete (L : in out LIST) is
-- Delete the current list element.
  Precurrent : LISTNODEPTR;
begin
    Precurrent := L.Current; -- Relink list to exclude
    L.Current.Left.Right := L.Current.Right;
    L.Current.Right.Left := L.Current.Left;
    if Precurrent.Right = L.Head  -- Tail node to be deleted?
    then L.CurrentPosition := 1;  -- Yes.
    end if;
    L.Current := L.Current.Right;
    if L.CurrentSize = 1
    then L.Head := null;
    else if L.Head = Precurrent
        then L.Head := L.Current;
        end if;
    end if;
    L.CurrentSize := L.CurrentSize -1;
    Precurrent := null;
end Delete;

procedure Create (L : in out LIST) is
-- Create list L
```

```
  begin
    L := new LISTINSTANCE;
    L.CurrentSize := 0;
    L.Head := null;
  end Create;

  procedure Kill (L : in out LIST) is
  -- Terminate list L
      P, Q : LISTNODEPTR;
      I    : natural := 1;
  begin
      P := L.Head;        -- Dispose list nodes
      loop
        exit when I >= L.CurrentSize;
        Q := P;
        P := P.Right;
        Q := null;
        I := I + 1;
      end loop;
    L := null;
  end Kill;

end LISTS;
```

```
-- This package is the N_ary tree data structure
-- used by the Static Scheduler.

with LISTS, TEXT_IO;
generic
 type NARY_TOKEN is private;

package N_ARY_TREE is
 type NARY_TREE is private;

 function IsEmpty (T : NARY_TREE) return BOOLEAN;
  -- Checks to see if the tree, 'T', has no nodes in it.
  -- Returns true if there are no nodes in the tree.
  -- Returns false if the tree has at least one defined node in it.

 procedure InsertRootNode (T  : in out NARY_TREE; El : in NARY_TOKEN);
  -- Creates a root node and places 'El' into the
  -- root's data field. If 'T' already has a root node
  -- defined then an error is generated.

 procedure InsertSiblingBefore (T  : in out NARY_TREE;
                                El : in NARY_TOKEN);
  -- Creates a new sibling node immediately prior to
  -- the Current node in the tree and inserts 'El' into
  -- the new node.  The Current node is updated to the newly inserted node.
  -- An attempt to add a sibling at the root node  will produce an error.

 procedure InsertChildBefore (T  : in out NARY_TREE;
                              El : in NARY_TOKEN);
  -- Places a new child node at the beginning of the Current node's child list.

 procedure UpdateNode (T  : in out NARY_TREE; El : in NARY_TOKEN);
  -- Replaces the data value of the Current node
  -- with value of 'El'.
  -- Produces an error when an attempt is made to update an empty tree.

 procedure FindChild (T  : in out  NARY_TREE; i  : in natural;
                      FOUND : out BOOLEAN);
  -- Locates the i'th child of the Current node.
  -- FOUND returns true if the Current node has i or more children.
  -- Current node will be the i'th child;
  -- FOUND returns false if the Current node has less than i children.
  -- Current node will be unchanged.
```

```
  procedure  FindParent (T : in  out NARY_TREE; FOUND : out BOOLEAN);
   -- FOUND returns true if Current node has
   -- a parent and Current node becomes the parent.
   -- FOUND returns false if the Current node is the
   -- root node and the Current node remains unchanged.

  procedure InsertSiblingAfter (T : in out NARY_TREE;
                                El : in NARY_TOKEN);
   -- Creates a new sibling node immediately after
   -- the Current node in the tree and inserts El
   -- into the new node.  The Current node is updated to
   -- the newly inserted node.

  procedure FindNext (T : in out NARY_TREE);
   -- Positions T.Current at the sibling immediately to the right of the Current node.
   --  An error is produced if the current node is the last sibling in the list.

  procedure DeleteChild (T : in out NARY_TREE; i : in natural);
   -- Deletes the ith child of the Current node and all of that child's descendants.

  procedure DeleteNode (T : in out NARY_TREE);
   -- Deletes the current node and all of its descendants.

  function NumChildren (T : in NARY_TREE) return NATURAL;
   -- returns the number of children in the current node.

  procedure FindRoot (T : in out NARY_TREE);
   -- Finds the root of the tree.

  procedure RetreiveNode (T  : in out NARY_TREE;   El : out NARY_TOKEN);
   --  Retreives the data value of Current node into 'El'.
   --  Produces an error when an attempt is made to retrieve from an empty tree.

private

  type NODE;
  type NodePtr is access NODE;
  package TREELIST is new LISTS(NodePtr);
  type NODE is
   record
     DATA     : NARY_TOKEN;
     CHILDLIST : TREELIST.LIST;
```

```
    PARENT   : NodePtr;
   end record;
 type NARY_TREE is
   record
    Root  : NodePtr;
    Current : NodePtr;
   end record;

end N_ARY_TREE;


package body N_ARY_TREE is

ERR1, ERR2, ERR3, ERR4, ERR5   : exception;

--procedure CreateMCTree (T : NARY_TREE);
--  Initializes the values of T to nil.
--begin   CreateMCTree
--  T.Root := nil;
--  T.Current := nil;
--end;   CreateMCTree

function IsEmpty (T : NARY_TREE) return BOOLEAN is
-- Checks to see if the tree, 'T', has no nodes in it.
-- Returns true if there are no nodes in the tree.
-- Returns false if the tree has at least one defined node in it.
begin
  return  T.Root = null;
end IsEmpty;

procedure InsertRootNode (T  : in out NARY_TREE; El : in NARY_TOKEN) is
--  Creates a root node and places 'El' into the root's data value.
--  If 'T' has already been defined then an error is generated.
begin
  if not IsEmpty (T) then
    raise ERR1;
  end if;
-- Tree is empty so go ahead and create new root node.
  T.ROOT := new NODE;
  T.Root.Parent := null;
  T.Root.Data := El;
  TREELIST.Create (T.Root.ChildList);
  T.Current := T.Root;
```

```
exception
 when ERR1 =>
   TEXT_IO.PUT_LINE
   ("Error: Attempting to create new root node in a non-empty tree.");
   TEXT_IO.SKIP_LINE;
end InsertRootNode;

procedure InsertSiblingBefore (T  : in out NARY_TREE;
                               El : in NARY_TOKEN) is
-- Creates a new sibling node immedRately prior to
-- the Current node in the tree and inserts 'El' into
-- the new node.  The Current node is updated to the  newly inserted node.
  Sibling   : NodePtr;
  POS       : NATURAL; -- Position of current node in its parent's Childlist
begin
 if IsEmpty (T) then
    InsertRootNode (T, El);
 else
    if T.Current = T.Root then
    --  Root of tree has no sibling!
      raise ERR2;
    end if;
-- normal sibling insertion
-- Locate position of current node in its parent's childlist.
     TREELIST.FindPosition(T.Current.Parent.ChildList, T.Current, POS);
-- Create new sibling node and insert 'El' into it
    Sibling := new NODE;
    Sibling.Data := El;
    TREELIST.Create (Sibling.ChildList);
    Sibling.Parent := T.Current.Parent;
     TREELIST.InsertBefore(T.Current.Parent.ChildList,Sibling);
    T.Current := Sibling;
 end if;
exception
 when ERR2 =>
   TEXT_IO.PUT_LINE ("Error: Cannot insert sibling at root of tree.");
   TEXT_IO.SKIP_LINE;
end  InsertSiblingBefore;

procedure InsertChildBefore (T  : in out NARY_TREE;
                             El : in NARY_TOKEN) is
--  Places a new child node at the beginning of the Current node's child list.
  Child  :  NodePtr;
```

```
begin
 if IsEmpty (T)  then
   InsertRootNode (T, El);
 else
   Child := new NODE;
   Child.Data := El;
   Child.Parent := T.Current;
   TREELIST.Create (Child.ChildList);
   if TREELIST.Size (T.Current.ChildList) > 0 then
     TREELIST.FindIth (T.Current.ChildList, 1);
   end if;
   TREELIST.InsertBefore (T.Current.ChildList, Child);
 end if;
end InsertChildBefore;

procedure UpdateNode (T  : in out NARY_TREE; El : in NARY_TOKEN) is
--  Replaces the data value of the Current node with value of 'El'.
begin
 if IsEmpty(T) then
  raise ERR3;
 end if;
-- Tree is not empty so update normally.
 T.Current.Data := El;
exception
 when ERR3 =>
   TEXT_IO.PUT_LINE ("Error: Attempting to update an empty tree.");
   TEXT_IO.SKIP_LINE;
end UpdateNode;

procedure FindChild (T      : in out NARY_TREE; i : in natural;
                     FOUND  : out BOOLEAN) is
-- Locates the i'th child of the Current node.
-- Returns true if the Current node has an i'th child.
-- Current node will be the i'th child;
-- Returns false if the Current node does not have an i'th child.
-- Current node will be unchanged.
begin
 if IsEmpty (T) then
   raise ERR4;
 end if;
-- tree is not empty
 if i > TREELIST.Size (T.Current.ChildList) then
   FOUND := false;
```

```
 else
    TREELIST.FindIth (T.Current.ChildList, i);
    TREELIST.Retrieve (T.Current.ChildList, T.Current);
   FOUND := true;
 end if;
exception
 when ERR4 =>
    TEXT_IO.PUT_LINE ("Error:  Attempting to find child in an empty tree.");
    TEXT_IO.SKIP_LINE;
end  FindChild;

procedure  FindParent (T : in out NARY_TREE; FOUND : out BOOLEAN) is
--  Makes the parent of the Current node, the Current node
--  Returns true if Current node has a parent and Current node becomes the parent.
--  Returns false if the Current node is the root node
--  and the Current node is unchanged.
begin
 if IsEmpty (T) then
   raise ERR5;
 end if;
-- Tree is not empty.
 if T.Current.Parent = null then
   FOUND := false;
 else
    T.Current := T.Current.Parent;
   FOUND :=  true;
 end if;
exception
 when ERR5 =>
    TEXT_IO.PUT_LINE ("Error: Attempting to find a  parent in an empty tree.");
    TEXT_IO.SKIP_LINE;
end FindParent;

procedure InsertSiblingAfter (T : in out NARY_TREE;
                              El : in NARY_TOKEN) is
-- Creates a new sibling node immediately after the
-- Current node in the tree and inserts El into the
-- new node.  The Current node is updated to
-- the newly inserted node.
  Sibling : NodePtr; -- The new sibling added
  FOUND : BOOLEAN;   -- records status of FindParent
  POS   : NATURAL;   -- records position of current node in its parent's childlist.
begin
```

```
   TempT.Current := T.Current;
   FindParent(T, FOUND);
   if FOUND then
      TREELIST.FindPosition (T.Current.ChildList, TempT.Current,POS);
   -- Create new sibling and initialize it before inserting
     Sibling := new NODE;
     Sibling.Data := El;
     TREELIST.Create (Sibling.ChildList);
     Sibling.Parent := TempT.Current.Parent;
     TREELIST.InsertAfter (T.Current.ChildList, Sibling);
    -- Update Current to point at new node in tree.
     T.Current := Sibling;
    else  -- Current node is root node.
     raise ERR2;
    end if;
   exception
    when ERR2 =>
      TEXT_IO.PUT_LINE ("Error: Cannot insert sibling  at root of tree.");
      TEXT_IO.SKIP_LINE;
end InsertSiblingAfter;

procedure FindNext (T : in out NARY_TREE) is
-- Positions T.Current at the sibling immediately to
-- the right of the Current node.
   TempT : NARY_TREE; -- Remembers position in tree upon entry
   FOUND : BOOLEAN;   -- Records status of FindParent operation
   LIST_END   : exception;
   POS        : NATURAL;
begin
   TempT.Current := T.Current;
   FindParent(T, FOUND);
   if FOUND then
      TREELIST.FindPosition(T.Current.ChildList,TempT.Current,POS);
      TREELIST.FindIth(T.Current.ChildList, POS + 1);
      TREELIST.Retrieve (T.Current.ChildList, T.Current);
   else  -- Current node is the root node.
     raise LIST_END;
   end if;
   exception
     when LIST_END =>
       TEXT_IO.PUT_LINE("Error: Cannot find next sibling.");
end FindNext;
```

```
procedure DeleteChild (T : in out NARY_TREE; i : in natural) is
-- Deletes the ith child of the T.Current node.
  NumKidsToDelete : natural;
  TempChild : TREELIST.LIST;
  TreeToDelete : NARY_TREE;
begin
   TREELIST.FindIth (T.Current.ChildList, i);
   TREELIST.Retrieve (T.Current.ChildList,
                      TreeToDelete.Current);
   NumKidsToDelete := TREELIST.Size(TreeToDelete.Current.ChildList);
   for j in 1 .. NumKidsToDelete loop
      DeleteChild (TreeToDelete, j);
   end loop;
   TREELIST.Kill (TreeToDelete.Current.ChildList);
   TreeToDelete.Current := null;
end DeleteChild;

procedure DeleteNode (T : in out NARY_TREE) is
-- Deletes sub tree rooted at T.Current
   TempT : NARY_TREE;  --Remembers position in tree upon entry
   FOUND  : BOOLEAN;   -- Records status of FindParent operation
   POS    : NATURAL;
begin
   TempT.Current := T.Current;
   FindParent (T, FOUND);
   if FOUND then
      TREELIST.FindPosition(T.Current.Childlist, TempT.Current,POS);
      DeleteChild(T,POS);
   else  -- Delete Root node, ie. delete entire tree.
      for j in 1 .. TREELIST.Size(T.Current.ChildList) loop
         DeleteChild(T,j);
      end loop;
      -- Now delete the root node itself.
      T.Root := null;
      T.Current := null;
   end if;
end DeleteNode;

function NumChildren (T : in NARY_TREE) return NATURAL is
begin
  return TREELIST.Size(T.Current.ChildList);
end NumChildren;
```

```
procedure FindRoot (T : in out NARY_TREE) is
begin
  T.Current := T.Root;
end FindRoot;

procedure RetreiveNode (T  : in out NARY_TREE; El : out NARY_TOKEN) is
--  Retreives the data value of the Current node into 'El'.
begin
  if IsEmpty(T) then
   raise ERR3;
  end if;
-- Tree is not empty so update normally.
  El := T.Current.Data;
exception
  when ERR3 =>
    TEXT_IO.PUT_LINE ("Error: Attempting to retreive from an empty tree.");
    TEXT_IO.SKIP_LINE;
end RetreiveNode;

end N_ARY_TREE;
```

APPENDIX D. STATIC SCHEDULER SOURCE CODE

This is the Ada source code for the packages FILES, FILE_PROCESSOR, and TOPOLOGICAL_SORTER of the Static Scheduler.

```
with VSTRINGS;
with Generic_List;
with N_ARY_TREE;
package FILES is
 package VARSTRING is new VSTRINGS(80);
 use VARSTRING;
 subtype DATA_STREAM is VSTRING;
 subtype OPERATOR_ID is VSTRING;
 subtype VALUE is NATURAL;
 subtype MET is VALUE;
 subtype MRT is VALUE;
 subtype MCP is VALUE;
 subtype PERIOD is VALUE;
 subtype WITHIN is VALUE;
 subtype STARTS is VALUE;
 subtype STOPS is VALUE;
 subtype LOWERS is VALUE;
 subtype UPPERS is VALUE;

 type LINKS is
  record
   THE_DATA_STREAM : DATA_STREAM;
   THE_FIRST_OP_ID : OPERATOR_ID;
   THE_LINK_MET    : MET := 0;
   THE_SECOND_OP_ID : OPERATOR_ID;
  end record;

 package LINKS_LIST is new GENERIC_LIST(LINKS);

 type OPERATORS is
  record
   THE_OPERATOR_ID : OPERATOR_ID;
   THE_MET         : MET := 0;
   THE_MRT         : MRT := 0;
```

```
    THE_MCP       : MCP  := 0;
    THE_PERIOD    : PERIOD  := 0;
    THE_WITHIN    : WITHIN  := 0;
   end record;

  package OPERATORS_LIST is new N_ARY_TREE(OPERATORS);

  package ATOMIC_LIST is new GENERIC_LIST(OPERATORS);

  type PRECEDENCE is
   record
    THE_LEFT_OP_ID  :  OPERATOR_ID;
    THE_RIGHT_OP_ID :  OPERATOR_ID;
   end record;

  package PRECEDENCE_LIST is new GENERIC_LIST(PRECEDENCE);

  type SCHEDULE_INPUTS is
   record
    OPERATOR       : OPERATOR_ID;
    THE_START      : STARTS  := 0;
    THE_STOP       : STOPS   := 0;
    THE_LOWER      : LOWERS  := 0;
    THE_UPPER      : UPPERS  := 0;
   end record;

  package SCHEDULE_INPUTS_LIST is new
            GENERIC_LIST(SCHEDULE_INPUTS);
end FILES;
```

```
with TEXT_IO;
with FILES;
package FILE_PROCESSOR is
  use FILES;

  procedure SEPARATE_DATA (LNKS : in out LINKS_LIST.LIST;
                           OPS  : in out OPERATORS_LIST.NARY_TREE);

  procedure VALIDATE_DATA(OPS: in out OPERATORS_LIST.NARY_TREE;
                          ATOMIC_OPS : in out ATOMIC_LIST.LIST);
  CRIT_OP_LACKS_MET          : exception;
  MET_REQUIRED               : exception;
  MET_GT_PARENT              : exception;
  MET_SUM_GT_PARENT          : exception;
  MET_NOT_LESS_THAN_PERIOD   : exception;
  MET_NOT_LESS_THAN_MRT      : exception;

end FILE_PROCESSOR;


with TEXT_IO;
with FILES;
use FILES;
with Generic_List;
package body FILE_PROCESSOR  is

  procedure SEPARATE_DATA (LNKS : in out LINKS_LIST.LIST;
                           OPS  : in out OPERATORS_LIST.NARY_TREE) is
  -- This procedure reads the output file from the
  --  Static Scheduler Attibute Grammer Processor
  --  and depending upon the key words that are
  --  declared as constants, separates the information
  --  in the file and stores it in one of three data
  --  structures.  These are OPS, which is a tree
  --  containing operator information, LNKS, which is a list of link statements,
  --  and St_List, which is a list of state variables.

    package STATES_LIST is new Generic_List(VARSTRING.VSTRING);

    MET    : constant VARSTRING.VSTRING := VARSTRING.VSTR("MET");
    MRT    : constant VARSTRING.VSTRING := VARSTRING.VSTR("MRT");
    MCP    : constant VARSTRING.VSTRING := VARSTRING.VSTR("MCP");
    PERIOD : constant VARSTRING.VSTRING :=
```

```
                                        VARSTRING.VSTR("PERIOD");
   WITHIN  : constant VARSTRING.VSTRING :=
                                        VARSTRING.VSTR("WITHIN");
   LINK : constant VARSTRING.VSTRING := VARSTRING.VSTR("LINK");
   STATE   : constant VARSTRING.VSTRING :=
                                        VARSTRING.VSTR("STATE");
   ENDSTATE: constant VARSTRING.VSTRING :=
                                          VARSTRING.VSTR("ENDSTATE");
   LINEAGE : constant VARSTRING.VSTRING :=
                                         VARSTRING.VSTR("LINEAGE");
   ATOMIC  : constant VARSTRING.VSTRING :=
                                        VARSTRING.VSTR("ATOMIC");
   ENDLINEAGE : constant VARSTRING.VSTRING :=
                                       VARSTRING.VSTR("END LINEAGE");
   Num_Links              : NATURAL := 1;
   Num_Sts                : NATURAL := 1;
   Current_Value          : Value;
   New_Word               : VARSTRING.VString;
   Current_Operator       : VARSTRING.VString;
   St_Name                : VARSTRING.VString;
   Cur_Op                 : OPERATORS;
   New_Op                 : OPERATORS;
   Cur_LINK               : LINKS;
   OpFound                : BOOLEAN := FALSE;
   DataStream_in_StList   : BOOLEAN := FALSE;
   St_List                : STATES_LIST.LIST;
   AG_OUTFILE             : TEXT_IO.FILE_TYPE;
   INPUT                  : TEXT_IO.FILE_MODE := TEXT_IO.IN_FILE;

   procedure FindOperator (OPS : in out OPERATORS_LIST.NARY_TREE;
                           OperatorName: in VARSTRING.VSTRING;
                           OpFound: in out BOOLEAN) is
   -- This procedure is used to find an operator
   -- in the tree of operators given the operator name.

    ChildFound  : BOOLEAN := False;
    CurrentOp   : OPERATORS;
    ParentFound : BOOLEAN := False;

   begin
    OPERATORS_LIST.RetreiveNode(OPS,CurrentOp);
      if VARSTRING.equal(CurrentOp.THE_OPERATOR_ID, OperatorName)
then
```

```
    OpFound := TRUE; -- The root contains OperatorName
   elsif OPERATORS_LIST.NumChildren(OPS) /= 0 then
    for Count in 1 .. OPERATORS_LIST.NumChildren(OPS)
    loop
      OPERATORS_LIST.FindChild(OPS,Count,ChildFound);
      OPERATORS_LIST.RetreiveNode(OPS,CurrentOp);
      if VARSTRING.equal(CurrentOp.THE_OPERATOR_ID,
                                  OperatorName) then
       OpFound := TRUE;
       exit;
      else
        FindOperator(OPS,OperatorName,OpFound);
      end if;
      exit when OpFound;
      OPERATORS_LIST.FindParent(OPS,ParentFound);
    end loop;
  end if;
 end FindOperator;

procedure TraverseOps (OPS : in out OPERATORS_LIST.NARY_TREE) is
-- This procedure is used to print the operators in the tree
 ChildFound : BOOLEAN := False;
 CurrentOp : OPERATORS;
 ParentFound : BOOLEAN := False;

begin
 OPERATORS_LIST.RetreiveNode(OPS, CurrentOp);
 VARSTRING.PUT_LINE(CurrentOP.THE_OPERATOR_ID);
 if OPERATORS_LIST.NumChildren(OPS) /= 0 then
  for Count in 1 .. OPERATORS_LIST.NumChildren(OPS)
  loop
    OPERATORS_LIST.FindChild(OPS,Count,ChildFound);
    TraverseOps(OPS);
    OPERATORS_LIST.FindParent(OPS,ParentFound);
  end loop;
 end if;
end TraverseOps;

begin
  TEXT_IO.OPEN (AG_OUTFILE, INPUT, "operator.info");
 while not TEXT_IO.END_OF_FILE(AG_OUTFILE) loop
   VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
   -- if the keyword is STATE
```

```
if VARSTRING.equal (New_Word,STATE) then
  while VARSTRING.notequal(New_Word,ENDSTATE)
 loop                -- build St_List
    VARSTRING.GET_LINE(AG_OUTFILE,New_Word);
    if VARSTRING.notequal(New_Word,ENDSTATE)then
      STATES_LIST.Insert(St_List,Num_Sts,New_Word);
     Num_Sts := Num_Sts + 1;
   end if;
 end loop;

-- if the keyword is MET
elsif VARSTRING.equal (New_Word,MET) then
  VALUE_IO.Get(AG_OUTFILE,Current_Value);
  TEXT_IO.SKIP_LINE(AG_OUTFILE);
  OPERATORS_LIST.FindRoot(OPS);
   FindOperator(OPS,Current_Operator,OpFound);
  OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
  Cur_Op.THE_MET := Current_Value;
  OPERATORS_LIST.UpdateNode(OPS,Cur_OP);;

-- if the keyword is MRT
elsif VARSTRING.equal (New_Word,MRT) then
   VALUE_IO.Get(AG_OUTFILE,Current_Value);
  TEXT_IO.SKIP_LINE(AG_OUTFILE);
  OPERATORS_LIST.FindRoot(OPS);
   FindOperator(OPS,Current_Operator,OpFound);
  OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
  Cur_Op.THE_MRT := Current_Value;
  OPERATORS_LIST.UpdateNode(OPS,Cur_OP);

-- if the keyword is MCP
elsif VARSTRING.equal (New_Word,MCP) then
  VALUE_IO.Get(AG_OUTFILE,Current_Value);
  TEXT_IO.SKIP_LINE(AG_OUTFILE);
  OPERATORS_LIST.FindRoot(OPS);
   FindOperator(OPS,Current_Operator,OpFound);
  OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
  Cur_Op.THE_MCP := Current_Value;
  OPERATORS_LIST.UpdateNode(OPS,Cur_OP);

-- if the keyword is PERIOD
elsif VARSTRING.equal (New_Word,PERIOD) then
  VALUE_IO.Get(AG_OUTFILE,Current_Value);
```

```
  TEXT_IO.SKIP_LINE(AG_OUTFILE);
  OPERATORS_LIST.FindRoot(OPS);
   FindOperator(OPS,Current_Operator,OpFound);
  OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
  Cur_Op.THE_PERIOD := Current_Value;
  OPERATORS_LIST.UpdateNode(OPS,Cur_OP);

-- if the keyword is WITHIN
elsif VARSTRING.equal (New_Word,WITHIN) then
   VALUE_IO.Get(AG_OUTFILE,Current_Value);
   TEXT_IO.SKIP_LINE(AG_OUTFILE);
   OPERATORS_LIST.FindRoot(OPS);
    FindOperator(OPS,Current_Operator,OpFound);
   OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
   Cur_Op.THE_WITHIN := Current_Value;
   OPERATORS_LIST.UpdateNode(OPS,Cur_OP);

-- if the keyword is LINK
 elsif VARSTRING.equal (New_Word,LINK) then
   VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
   STATES_LIST.Length(St_List,Num_Sts);
   for Count in 1..Num_Sts
   loop   -- look for the data stream in St_List
     STATES_LIST.Find_Item(St_List,Count,St_Name);
     if VARSTRING.equal(St_Name,New_Word) then
       DataStream_in_StList := TRUE;
      exit;
     end if;
   end loop;
   if DataStream_in_StList then  -- discard link stmt
     VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
     VALUE_IO.GET(AG_OUTFILE,Current_Value);
     TEXT_IO.SKIP_LINE(AG_OUTFILE);
     VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
     DataStream_in_StList := FALSE;
   else                    -- store link stmt
     Cur_Link.THE_DATA_STREAM := New_Word;
     VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
     Cur_Link.THE_FIRST_OP_ID := New_Word;
      VALUE_IO.GET(AG_OUTFILE,Current_Value);
     TEXT_IO.SKIP_LINE(AG_OUTFILE);
     Cur_Link.THE_LINK_MET := Current_Value;
     VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
```

```
      Cur_Link.THE_SECOND_OP_ID := New_Word;
      LINKS_LIST.Insert(LNKS,Num_Links,Cur_Link);
      Num_Links := Num_Links + 1;
    end if;

   -- if the keyword is LINEAGE
   elsif VARSTRING.equal(New_Word,LINEAGE) then
    VARSTRING.GET_LINE(AG_OUTFILE,New_Word);
    OPERATORS_LIST.FindRoot(OPS);
    if OPERATORS_LIST.IsEmpty(OPS) then
     New_Op.THE_OPERATOR_ID := New_Word;
     OPERATORS_LIST.InsertRootNode(OPS,New_Op);
     while VARSTRING.notequal(New_Word,ENDLINEAGE)
     loop  -- insert root's children
      VARSTRING.GET_LINE(AG_OUTFILE,New_Word);
      if (VARSTRING.notequal(New_WORD,ENDLINEAGE)) and
        (VARSTRING.notequal(New_WORD,ATOMIC))  then
        New_Op.THE_OPERATOR_ID := New_Word;
        OPERATORS_LIST.InsertChildBefore(OPS,New_Op);
      end if;
     end loop;
    else  -- find parent operator
     FindOperator(OPS,New_Word,OpFound);
     OPERATORS_LIST.RetreiveNode(OPS,Cur_OP);
     while VARSTRING.notequal(New_Word,ENDLINEAGE)
     loop
      VARSTRING.GET_LINE(AG_OUTFILE,New_Word);
      if (VARSTRING.notequal(New_WORD,ENDLINEAGE)) and
        (VARSTRING.notequal(New_WORD,ATOMIC))  then
        New_Op.THE_OPERATOR_ID := New_Word;
        OPERATORS_LIST.InsertChildBefore(OPS,New_Op);
      end if;
     end loop;
    end if;

    else  -- not a keyword, an operator name
     Current_Operator := New_Word;
   end if;
   OpFound := FALSE;
  end loop;

end SEPARATE_DATA;
```

```
procedure VALIDATE_DATA
                              (OPS : in out OPERATORS_LIST.NARY_TREE;
                               ATOMIC_OPS : in out ATOMIC_LIST.LIST) is

  subtype OPS_POINTER is OPERATORS_LIST.NARY_TREE;
  package MET_LIST is new GENERIC_LIST(OPS_POINTER);

  Ops_with_MET        : MET_LIST.LIST;
  This_Op             : OPS_POINTER;
  Cur_Op              : OPERATORS;
  Child_Op            : OPERATORS;
  Num_MET             : NATURAL := 1;
  NumAtomics          : NATURAL := 1;
  ChildNum            : NATURAL := 1;
  MET_Sum             : NATURAL := 0;
  ChildFound          : BOOLEAN := False;
  ParentFound         : BOOLEAN := False;
  NON_CRITS           : TEXT_IO.FILE_TYPE;
  OUTPUT              : TEXT_IO.FILE_MODE := TEXT_IO.OUT_FILE;

procedure CheckTiming (OPS  : in out OPERATORS_LIST.NARY_TREE) is
-- This procedure is used to find operators without an
--   MET that have other timing constraints. This is not allowed.

  ChildFound        : BOOLEAN := False;
  CurrentOp         : OPERATORS;
  ParentFound       : BOOLEAN := False;

begin
  OPERATORS_LIST.RetreiveNode(OPS,CurrentOp);
  if CurrentOp.THE_MET = 0 then
    if CurrentOp.THE_MRT /= 0 or else CurrentOP.THE_MCP /= 0 or else
      CurrentOP.THE_PERIOD /= 0 or else CurrentOP.THE_WITHIN /= 0 then
        Exception_Operator := CurrentOp.THE_OPERATOR_ID;
        raise CRIT_OP_LACKS_MET;
    end if;
  end if;
  if OPERATORS_LIST.NumChildren(OPS) /= 0 then
   for Count in 1 .. OPERATORS_LIST.NumChildren(OPS)
   loop
     OPERATORS_LIST.FindChild(OPS,Count,ChildFound);
     CheckTiming(OPS);
     OPERATORS_LIST.FindParent(OPS,ParentFound);
```

```
    end loop;
  end if;
end CheckTiming;

procedure StoreOps (OPS  : in out OPERATORS_LIST.NARY_TREE;
                    Ops_with_MET : in out MET_LIST.LIST) is
-- This procedure is used to find operators that
--   have a MET in the tree and store them in a list

  ChildFound  : BOOLEAN := False;
  CurrentOp   : OPERATORS;
  ParentFound : BOOLEAN := False;

begin
  OPERATORS_LIST.RetreiveNode(OPS,CurrentOp);
  if CurrentOp.THE_MET /= 0 then
    This_Op := OPS;
    MET_LIST.Insert(Ops_with_MET,Num_MET,This_Op);
    Num_MET := Num_MET + 1;
  end if;
  if OPERATORS_LIST.NumChildren(OPS) /= 0 then
   for Count in 1 .. OPERATORS_LIST.NumChildren(OPS)
   loop
     OPERATORS_LIST.FindChild(OPS,Count,ChildFound);
     StoreOps(OPS,Ops_with_MET);
     OPERATORS_LIST.FindParent(OPS,ParentFound);
   end loop;
  end if;
end StoreOps;

procedure SortAtomics
        (OPS  : in out OPERATORS_LIST.NARY_TREE;
         ATOMIC_OPS: in out ATOMIC_LIST.LIST) is
-- This procedure is used to find the atomic operators
--   and send those without METs to the NON_CRITS file
--   and insert those with METs into a list of
--   critical atomic operators for further processing

  ChildFound      : BOOLEAN := False;
  CurrentOp       : OPERATORS;
  ParentFound     : BOOLEAN := False;

begin
```

```
 if OPERATORS_LIST.NumChildren(OPS) = 0 then
   -- it is an atomic Op
   OPERATORS_LIST.RetreiveNode(OPS, CurrentOp);
   if CurrentOp.THE_MET = 0 then -- put in NON_CRITS
     VARSTRING.PUT_LINE(NON_CRITS,CurrentOp.THE_OPERATOR_ID);
   elsif CurrentOp.THE_MET /= 0 then
    -- store in Atomic_Ops
    ATOMIC_LIST.Insert(Atomic_Ops,NumAtomics,CurrentOp);
    NumAtomics := NumAtomics + 1;
   end if;
 elsif OPERATORS_LIST.NumChildren(OPS) /= 0 then
  for Count in 1 .. OPERATORS_LIST.NumChildren(OPS)
  loop
    OPERATORS_LIST.FindChild(OPS,Count,ChildFound);
    SortAtomics(OPS,Atomic_Ops);
    OPERATORS_LIST.FindParent(OPS,ParentFound);
  end loop;
 end if;
end SortAtomics;

begin
 OPERATORS_LIST.FindRoot(OPS);
 CheckTiming(OPS); -- ensure ops with timing have METs
 OPERATORS_LIST.FindRoot(OPS);
 StoreOps(OPS,Ops_with_MET); -- collect ops with METs
 MET_LIST.Length(Ops_with_MET,Num_MET);
 for Count in 1.. Num_MET
  MET_LIST.Find_Item(Ops_with_MET,Count,This_Op);
  OPERATORS_LIST.RetreiveNode(This_Op,Cur_Op);
  if OPERATORS_LIST.NumChildren(This_Op) /= 0 then
   for ChildCount in 1..OPERATORS_LIST.NumChildren(This_Op)
   loop
     OPERATORS_LIST.FindChild(This_Op,ChildCount,ChildFound);
     OPERATORS_LIST.RetreiveNode(This_Op,Child_Op);
     MET_Sum := MET_Sum + Child_Op.THE_MET;
     if Child_Op.THE_MET = 0 then
       Exception_Operator := Child_OP.THE_OPERATOR_ID;
       raise MET_REQUIRED;
     elsif Cur_Op.THE_MET /= 0 then
       if Child_Op.THE_MET > Cur_OP.THE_MET then
         Exception_Operator := Child_OP.THE_OPERATOR_ID;
         raise MET_GT_PARENT;
       end if;
```

```
      end if;
      OPERATORS_LIST.FindParent(This_Op,ParentFound);
    end loop;
    if MET_Sum > Cur_Op.THE_MET then
      Exception_Operator := Cur_OP.THE_OPERATOR_ID;
      raise MET_SUM_GT_PARENT;
    end if;
    MET_Sum := 0;
   end if;
  end loop;

 -- Separate the non-time critical atomic operators
 --   from the time critical operators.
  TEXT_IO.CREATE(NON_CRITS,OUTPUT,"NON_CRITS");
  OPERATORS_LIST.FindRoot(OPS);
-- collect atomic operators with METs
-- send atomic operators without METs to NON_CRITS
  SortAtomics(OPS,Atomic_Ops);
  TEXT_IO.CLOSE(NON_CRITS);
  ATOMIC_LIST.Length(Atomic_Ops,NumAtomics);
  for Count in 1..NumAtomics
  loop
    ATOMIC_LIST.Find_Item(Atomic_Ops,Count,Cur_Op);
    -- Check to ensure that the PERIOD is greater than the MET.
    if Cur_Op.THE_PERIOD /=0 and then
      Cur_Op.THE_MET >= Cur_Op.THE_PERIOD then
      Exception_Operator := Cur_OP.THE_OPERATOR_ID;
      raise MET_NOT_LESS_THAN_PERIOD;
    end if;
    -- Check to ensure that the MRT is greater than the MET.
    if Cur_Op.THE_MRT /= 0 and then
      Cur_Op.THE_MET >= Cur_Op.THE_MRT then
       Exception_Operator := Cur_OP.THE_OPERATOR_ID;
       raise MET_NOT_LESS_THAN_MRT;
    end if;
   end loop;

 end VALIDATE_DATA;

end FILE_PROCESSOR;
```

```
with FILES;
use FILES;
with TEXT_IO;
package TOPOLOGICAL_SORTER is

  procedure CREATE_LISTS (LNKS  : in out LINKS_LIST.LIST;
                          PRECE : in out PRECEDENCE_LIST.LIST);
  procedure SORT_REMAINING_OPERATORS
                          (LNKS  : in out LINKS_LIST.LIST;
                           PRECE : in out PRECEDENCE_LIST.LIST);

  NO_INITIAL_LINK_OP     : exception;
  NO_MATCHES_FOUND       : exception;

end TOPOLOGICAL_SORTER;

with TEXT_IO;
with FILES;
with GENERIC_LIST;
package body TOPOLOGICAL_SORTER is
-- This package determines the precedence order in which operators must
--  execute in the final schedule.  This information is determined
--  from the links statements.

  procedure CREATE_LISTS (LNKS : in out LINKS_LIST.LIST;
                          PRECE: in out PRECEDENCE_LIST.LIST) is
  -- This procedure determines which operators in the
  --  links list must be executed before another other.
  --  These operators are identified by either having
  --  only EXTERNAL inputs and no other inputs, or
  --  by appearing as the first operator, but never
  --  appearing as the second operator in a link
  --  statement.  Once these operators are identified,
  --  the links statments are searched for them.  When
  --  found, the precedence list is searched
  --  to determine if they are already in it.  If not,
  --  they are inserted in the precedence list.
  EXTERNAL   : constant VARSTRING.VSTRING :=
                                VARSTRING.VSTR("EXTERNAL");
  Num_Links             : NATURAL;
  Num_Prece             : NATURAL;
  I_Link                : LINKS;
  J_Link                : LINKS;
```

```
Cur_Link                : LINKS;
First                    : BOOLEAN := TRUE;  -- first in precedence
Already_in_List          : BOOLEAN := FALSE; -- already in precedence list
Cur_Prece                : PRECEDENCE;
List_Prece               : PRECEDENCE;
Old_Prece                : PRECEDENCE;
Op_Name                  : VARSTRING.VSTRING;

begin
 LINKS_LIST.Length(LNKS,Num_Links);
 for I_Count in 1..Num_Links
 loop
  LINKS_LIST.Find_Item(LNKS,I_Count,I_Link);
  -- find an operator whose input comes from
  -- EXTERNAL and never appears as the second operator in a link statement
  if VARSTRING.equal(I_Link.THE_FIRST_OP_ID, EXTERNAL) then
   Op_Name := I_Link.THE_SECOND_OP_ID;
   for J_Count in 1..Num_Links
   loop
    LINKS_LIST.Find_Item(LNKS,J_Count,J_Link);
    if VARSTRING.equal(I_Link.THE_SECOND_OP_ID, and then
     VARSTRING.notequal(J_Link.THE_FIRST_OP_ID, EXTERNAL) then
      First := FALSE;
     exit;
    end if;
   end loop;
   -- find an operator who appears as the first operator in a link statement
   -- but never appears as the second operator
  else
   Op_Name := I_Link.THE_FIRST_OP_ID;
   for J_Count in 1..Num_Links
   loop
    LINKS_LIST.Find_Item(LNKS,J_Count,J_Link);
    if VARSTRING.equal(I_Link.THE_FIRST_OP_ID,
                                J_Link.THE_SECOND_OP_ID) then
      First := FALSE;
     exit;
    end if;
   end loop;
  end if;
  if First then
   -- find link statements where  THE_FIRST_OP_ID = OpName
   -- and store them in the precedence list
```

```
    for First_Count in 1..Num_Links
    loop
      LINKS_LIST.Find_Item(LNKS,First_Count,Cur_Link);
      if VARSTRING.equal(Cur_Link.THE_FIRST_OP_ID, Op_Name) then
        Cur_Prece.THE_LEFT_OP_ID := Cur_Link.THE_FIRST_OP_ID;
        Cur_Prece.THE_RIGHT_OP_ID := Cur_Link.THE_SECOND_OP_ID;
        -- determine if operators already in precedence list
        PRECEDENCE_LIST.Length(PRECE,Num_Prece);
        for Prece_Count in 1..Num_Prece
        loop
          PRECEDENCE_LIST.Find_Item(PRECE,Prece_Count,Old_Prece);
          if VARSTRING.equal(Cur_Prece.THE_LEFT_OP_ID,
                                  Old_Prece.THE_LEFT_OP_ID) and
            VARSTRING.equal(Cur_Prece.THE_RIGHT_OP_ID,
                                  Old_Prece.THE_RIGHT_OP_ID) then
            Already_in_List := TRUE;
           exit;
          end if;
        end loop;
        -- if not then insert in precedence list
        if not Already_in_List then
          PRECEDENCE_LIST.Insert(PRECE, Num_Prece+1, Cur_Prece);
        end if;
      end if;
      Already_in_List := FALSE;
    end loop;
   end if;
   First := TRUE;
  end loop;
  PRECEDENCE_LIST.Length(PRECE,Num_Prece);
  if Num_Prece = 0 then
    raise NO_INITIAL_LINK_OP;
  end if;
end CREATE_LISTS;

procedure SORT_REMAINING_OPERATORS
                              (LNKS : in out LINKS_LIST.LIST;
                                PRECE : in out PRECEDENCE_LIST.LIST) is

  Num_Prece              : NATURAL;
  New_Num_Prece          : NATURAL;
  Cur_Prece              : PRECEDENCE; -- precedence element
```

```
procedure SORT_OPERATORS (LNKS : in out LINKS_LIST.LIST;
                          PRECE : in out PRECEDENCE_LIST.LIST) is
-- This procedure is recursive and continues to go
--  through the precedence list and links list
--  searching for first operators in the links
--  list that matches right operators in the precedence
--  list.  Once found, that link statement is added
--  to the precedence list.  Execution is terminated
--  when a pass has been made through the precedence
--  list and no new elements have been added.
 EXTERNAL: constant VARSTRING.VSTRING :=
                                   VARSTRING.VSTR("EXTERNAL");
 Num_Links          : NATURAL;
 Num_Prece          : NATURAL;
 New_Num_Prece      : NATURAL;
 Cur_Prece          : PRECEDENCE; -- current element
 New_Prece          : PRECEDENCE; -- new element to be added
 Old_Prece          : PRECEDENCE; -- old element from list
 Cur_Link           : LINKS;
 Already_in_List    : BOOLEAN := FALSE;

 begin
   PRECEDENCE_LIST.Length(PRECE,Num_Prece);
   for PreceCount in 1.. Num_Prece
  loop
     PRECEDENCE_LIST.Find_Item(PRECE,PreceCount,Cur_Prece);
    LINKS_LIST.Length(LNKS,Num_Links);
    for LinkCount in 1.. Num_Links
    loop
      LINKS_LIST.Find_Item(LNKS,LinkCount,Cur_Link);
      -- find a precedence element whose right operator is not
      --  EXTERNAL and then whose left operator is the first
      --  operator in the links list
      if VARSTRING.notequal(Cur_Prece.THE_RIGHT_OP_ID,EXTERNAL)
        and then
         VARSTRING.equal(Cur_Prece.THE_RIGHT_OP_ID,
                         Cur_Link.THE_FIRST_OP_ID) then
        New_Prece.THE_LEFT_OP_ID := Cur_Link.THE_FIRST_OP_ID;
        New_Prece.THE_RIGHT_OP_ID := Cur_Link.THE_SECOND_OP_ID;
        -- determine if the links operators are already in the precedence list
        PRECEDENCE_LIST.Length(PRECE,New_Num_Prece);
        for Prece_Count in 1..New_Num_Prece
       loop
```

```
            PRECEDENCE_LIST.Find_Item(PRECE,Prece_Count, Old_Prece);
            if VARSTRING.equal(New_Prece.THE_LEFT_OP_ID,
                               Old_Prece.THE_LEFT_OP_ID) and
              VARSTRING.equal(New_Prece.THE_RIGHT_OP_ID,
                               Old_Prece.THE_RIGHT_OP_ID) then
              Already_in_List := TRUE;
             exit;
            end if;
           end loop;
           -- if not then insert them in precedence list
           if not Already_in_List then
             PRECEDENCE_LIST.Insert(PRECE,New_Num_Prece+1, New_Prece);
           end if;
           Already_in_List := FALSE;
         else
           Exception_Operator := Cur_Prece.THE_RIGHT_OP_ID;
         end if;
        end loop;
       end loop;
       PRECEDENCE_LIST.Length(PRECE,New_Num_Prece);
       if New_Num_Prece > Num_Prece then
         Sort_Operators(LNKS,PRECE);
       end if;
     end SORT_OPERATORS;

  begin
    PRECEDENCE_LIST.Length(PRECE,Num_Prece);
    SORT_OPERATORS(LNKS,PRECE);
    PRECEDENCE_LIST.Length(PRECE,New_Num_Prece);
    if Num_Prece = New_Num_Prece then
     raise NO_MATCHES_FOUND;
    end if;
  end SORT_REMAINING_OPERATORS;

end TOPOLOGICAL_SORTER
```

```
with FILES;
with FILE_PROCESSOR;
with TOPOLOGICAL_SORTER;
use FILES;
procedure Static_Scheduler is
-- This is the main driver for the Static Scheduler.  It
--  calls the procedures within the FILE_PROCESSOR and
--  TOPOLOGICAL_SORTER packages.  When complete it will
--  also call the procedures within HARMONIC_BLOCK_BUILDER
--  and OPERATOR_SCHEDULER.

 LNKS                 : LINKS_LIST.LIST;
 OPS                  : OPERATORS_LIST.NARY_TREE;
 ATOMIC_OPS           : ATOMIC_LIST.LIST;
 PRECE                : PRECEDENCE_LIST.LIST;
--  Remove the following variable for embedding in the Debugger.
 Exception_Operator : VARSTRING.VSTRING := VARSTRING.VSTR("");
begin
 FILE_PROCESSOR.SEPARATE_DATA(LNKS,OPS);
 FILE_PROCESSOR.VALIDATE_DATA(OPS,ATOMIC_OPS);
 TOPOLOGICAL_SORTER.CREATE_LISTS(LNKS,PRECE);
 TOPOLOGICAL_SORTER.SORT_REMAINING_OPERATORS(LNKS,
                                                    PRECE);

-- Remove the TEXT_IO lines and uncomment the SS_Debug lines
--  for embedding in the Static Scheduler Debugger.

 exception
  when FILE_PROCESSOR.CRIT_OP_LACKS_MET =>
   TEXT_IO.PUT_LINE("Critical operator must have an MET.");
   -- SS_Debug.CRIT_OP_LACKS_MET;
  when FILE_PROCESSOR.MET_REQUIRED =>
   TEXT_IO.PUT_LINE("Composite has MET.  MET required.");
   -- SS_Debug.MET_REQUIRED;
  when FILE_PROCESSOR.MET_GT_PARENT =>
   TEXT_IO.PUT_LINE("Operator's MET greater than parent's MET.");
   -- SS_Debug.MET_GT_PARENT;
  when FILE_PROCESSOR.MET_SUM_GT_PARENT =>
   TEXT_IO.PUT_LINE("MET sum greater than parent operator's MET.");
   -- SS_Debug.MET_SUM_GT_PARENT;
  when FILE_PROCESSOR.MET_NOT_LESS_THAN_MRT =>
   TEXT_IO.PUT_LINE("MET larger than or equal to MRT.");
   -- SS_Debug.MET_NOT_LESS_THAN_MRT;
```

```
   when FILE_PROCESSOR.MET_NOT_LESS_THAN_PERIOD =>
     TEXT_IO.PUT_LINE("MET greater than period.");
     -- SS_Debug.MET_NOT_LESS_THAN_PERIOD;
   when TOPOLOGICAL_SORTER.NO_INITIAL_LINK_OP =>
     TEXT_IO.PUT_LINE("Cannot locate the initial link statement.");
     -- SS_Debug.NO_INITIAL_LINK_OP;
   when TOPOLOGICAL_SORTER.NO_MATCHES_FOUND =>
     TEXT_IO.PUT_LINE("No link statements after the first.");
     -- SS_Debug.NO_MATCHES_FOUND;
end Static_Scheduler;
```

BIBLIOGRAPHY

Booch, Grady, *Software Engineering with Ada*, 2nd ed., Benjamin/Cummings, 1983.

Janson, D., and Luqi,"A Static Scheduler for the Computer Aided Prototyping System," *Proceedings of COMPASS*, pp. 92-98, June 1988.

Luqi, "Execution of Real-Time Prototypes," *ACM First International Workshop on Computer-Aided Software Engineering*, v.2, pp. 870-884, May 1987.

Luqi and Ketabchi, M, "A Computer Aided Prototyping System," *IEEE Software*, pp. 66-72, March 1988.

INITIAL DISTRIBUTION LIST

No. of Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Office of Naval Research 1
   Office of the Chief of Naval Research
   Attn. CDR Michael Gehl, Code 1224
   800 N. Quincy Street
   Arlington, VA 22217-5000

4. Space and Naval Warfare Systems Command 1
   Attn: Dr. Knudsen, Code PD 50
   Washington, D. C. 20363-5100

5. Ada Joint Program Office 1
   OUSDRE (R&AT)
   Pentagon
   Washington, D.C. 20301

6. Naval Sea Systems Command 1
   Attn: CAPT Joel Crandall
   National Center #2, Suite 7NO6
   Washington, D. C. 22202

7. Office of the Secretary of Defense 1
   Attn: CDR Barber
   STARS Program Office
   Washington, D. C. 20301

8. Office of the Secretary of Defense 1
   Attn: Mr. Joel Trimble
   STARS Program Office
   Washington, D. C. 20301

9. Commanding Officer 1
Naval Research Laboratory
Code 5150
Attn: Dr. Elizabeth Wald
Washington, D. C. 20375-5000

10. Navy Ocean System Center 1
Attn: Linwood Sutton, Code 423
San Diego, CA 92152-500

11. National Science Foundation 1
Attn: Dr. William Wulf
Washington, DC 20550

12. National Science Foundation 1
Division of Computer and Computation Research
Attn: Dr. Peter Freeman
Washington, D. C. 20550

13. National Science Foundation 1
Director, PYI Program
Attn: Dr. C. Tan
Washington, D. C. 20550

14. Office of Naval Research 1
Computer Science Division, Code 1133
Attn: Dr. Van Tilborg
800 N. Quincy Street
Arlington, VA 22217-5000

15. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn: Mr. J. Smith
800 N. Quincy Street
Arlington, VA 22217-5000

16. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn: Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, VA 22209-2308

17. Defense Advanced Research Projects Agency (DARPA)..... 1
Integrated Strategic Technology Office (ISTO)
Attn: Dr. Squires
1400 Wilson Boulevard
Arlington, VA 22209-2308

18. Defense Advanced Research Projects Agency (DARPA)..... 1
Integrated Strategic Technology Office (ISTO)
Attn: MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, VA 22209-2308

19. Defense Advanced Research Projects Agency (DARPA)..... 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, VA 22209-2308

20. Defense Advanced Research Projects Agency (DARPA)..... 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, VA 22209-2308

21. Defense Advanced Research Projects Agency (DARPA)..... 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, VA 22209-2308

22. Defense Advanced Research Projects Agency (DARPA)..... 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 22209-2308

23. COL C. Cox, USAF.....1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D. C. 20318-8000

24. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D. C. 20318-8000

25. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn: Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, NY 13441-5700

26. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn: Mr. William E. Rzepka
Griffis Air Force Base, NY 13441-5700

27. Professor Luqi 1
Code 52LQ
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943-5100

28. LCDR Laura C. Marlowe 1
FACSFAC, Bldg. 93
NAS North Island
San Diego, CA 92135

Thesis
M3475
c.1

Marlowe
A Static Scheduler for critical timing constraints.